White Paper

Modeling with IBM ILOG CPLEX CP Optimizer – Practical Scheduling Examples

IBM

Contents

Introduction

This paper demonstrates how to use IBM® ILOG® CPLEX® CP Optimizer to efficiently model and solve practical scheduling problems. It includes a brief introduction to CPLEX CP Optimizer, as well as an overview of the available modeling features. In addition, it includes a set of real-world scheduling examples chosen to demonstrate frequently used modeling patterns of practical costs and constraints.

CPLEX CP Optimizer can be used directly through Application Programming Interfaces (APIs) in Java™, C#, or C++, and through IBM ILOG CPLEX Optimization Studio. The examples in this paper use CPLEX Optimization Studio, and the included Optimization Programming Language (OPL).

About IBM ILOG CPLEX CP Optimizer

CPLEX CP Optimizer is a next-generation Constraint Programming (CP) system designed for an easy-to-use "model and run" approach. It efficiently addresses detailed scheduling problems, as well as certain combinatorial optimization problems that cannot be easily linearized and solved using traditional Mathematical Programming methods.

This paper focuses on using CPLEX CP Optimizer for detailed scheduling problems. A major benefit of the CPLEX CP Optimizer approach to scheduling is that no enumeration of time, i.e. time buckets or time periods, is required. This means that relatively few decision variables are needed compared to a Mathematical Programming (MP) approach that would require variables for each time bucket of a discretized model. Therefore, a very fine time granularity can be used to describe constraints and costs—models can be formulated and solved efficiently, regardless of whether the relevant unit of time to describe a scheduling problem is thought of in milliseconds, minutes, or hours.

IBM ILOG CPLEX CP Optimizer for detailed scheduling

CPLEX CP Optimizer addresses scheduling problems by assigning start and end times to intervals during which, for example, a task occurs, while effectively managing resource constraints over time and alternative modes to perform a task.

A typical scheduling problem is defined by:

- A set of time intervals representing activities, operations, or tasks to be completed, that might be optional or mandatory
- A set of constraints that apply during these intervals or between intervals
- An objective function to minimize or maximize, for example minimizing the time required to perform a set of tasks, maximizing the number of tasks successfully scheduled within a given horizon, or minimizing the cost of delivering tasks past their due date

CPLEX CP Optimizer as embedded in CPLEX Optimization Studio provides specialized variables, constraints and keywords designed for modeling scheduling problems. This set of features is both concise and expressive, allowing easy modeling of a wide range of scheduling problems, and is supported by a powerful solution engine.

The APIs of CPLEX CP Optimizer provide access to the same type of decision variables, constraints and expressions to build scheduling models, as those available through the OPL language. The examples in this paper are written in OPL.

Example summary

The application areas demonstrated in this paper are:

Oversubscribed scheduling: In this type of scheduling problem, the cumulative demand for resources exceeds their availability and the objective is to schedule as many tasks as possible given the constrained resources. This paper demonstrates modeling this type of problem using the United States Air Force (USAF) Satellite Control Network scheduling problem (Kramer et al., 2007).

Project scheduling: Here, a project is defined in terms of a hierarchical set of tasks (the work breakdown structure), with precedence constraints between tasks. The goal is to assign workers to tasks so as to minimize both the project makespan (the total time to complete the project) and the total worker cost, with some constraints involving the workers' proficiencies. You will start by building a basic model, and learn how to extend the model to incorporate constraints on workers' schedules, as well as the ability to cancel certain subprojects at a fixed cost.

Transportation scheduling: A number of truck drivers are available to be scheduled to complete a set of driving tasks. There are various constraints related to the shifts each driver is able to work, due to Hours-Of-Service (HOS) regulations specified by the Department of Transportation. The objective is to minimize the total number of shifts worked.

Personal task scheduling: The problem of personal task scheduling addressed here is based on an application described by Refanidis (2007). A number of tasks, some of which may be split into several parts, must be scheduled during a given time horizon. The problem characteristics to be addressed include precedence constraints, transition times, task preferences, and calendars.

Batch scheduling: A set of product orders must be scheduled on a number of finite capacity machines. Each product is manufactured using a unique process that consists of multiple steps. In this example, the processing steps are heat treatments characterized by a combination of temperature level and treatment duration. Oven transition times between processing steps are dependent on the temperature change between steps. In addition, a material preparation time is required before each step of the same product order. The objective is to minimize the makespan.

Each example consists of:

- A problem description
- An overview of the modeling features demonstrated
- A step-by-step guide to modeling the problem using CPLEX CP Optimizer
- One or more partially completed solutions
- The final solution

Try to model each step before looking at the solution. All solution files are written using IBM ILOG OPL.

Software requirements

You need to have CPLEX Optimization Studio (IBM ILOG OPL Development Studio 6.1 or later) installed to be able to run the models. A trial version of IBM ILOG CPLEX Optimization Studio, that also includes trial versions of the CPLEX Optimizers for Mathematical Programming and CPLEX CP Optimizer for Constraint Programming, is available at www.ibm.com/software/websphere/products/optimization/cplex-studio-preview-edition/

A small data set appropriate for trial mode is provided with each example, in addition to the complete data set. However, the examples are best followed using the full version of the software. To request a full evaluation version, please contact your IBM representative.

Prerequisites

This paper assumes some basic knowledge of either Mathematical Programming (MP) or Constraint Programming (CP), as well as some familiarity with IBM ILOG CPLEX Optimization Studio.

The following papers are recommended reading before continuing with the examples in this paper:

- *Detailed Scheduling in IBM ILOG CPLEX Optimization Studio with IBM ILOG CPLEX CP Optimizer:* An overview of the functionality available in IBM ILOG CPLEX CP Optimizer.
- *Efficient Modeling in IBM ILOG CPLEX Optimization Studio:* The OPL models you will be looking at make extensive use of an OPL modeling construct called a "tuple." This paper provides a summary of tuples and the benefits of using them.

Both these papers are available online at www.ibm.com/software/integration/optimization/cplex-cp-optimizer/

About IBM ILOG CPLEX Optimization Studio

CPLEX Optimization Studio supports the rapid development, deployment and maintenance of Mathematical Programming (MP) and Constraint Programming (CP) models from a powerful integrated development environment (IDE) built on the Optimization Programming Language (OPL), through programmatic APIs, or alternatively through third-party modeling environments.

OPL provides a natural representation of optimization models, requiring far less effort than general-purpose programming languages. Debugging and tuning tools support the development process, and once ready, the model can be deployed into an external application. OPL models can be easily integrated into any application written in Java, .NET or C++.

Additionally, CPLEX Optimization Studio is tightly integrated with IBM ILOG ODM Enterprise (optional), providing push-button generation of ODM Enterprise applications based on OPL models. A simple wizard-guided step produces an initial application, mapping OPL data structures to data tables in ODM Enterprise, decision variables and solution metrics to solution views, and objective functions to ODM Enterprise's interactive business goals. ODM Enterprise provides rapid development of analytical decision support applications for immediate deployment. It helps users to adjust assumptions, operating constraints and goals, and see the engine's recommendations in familiar business terminology. It also provides extensive support for what-if analysis, scenario comparison, solution explanations, and the controlled relaxation of binding constraints.

Modeling features overview

Figure 1 shows a summary of the modeling features for scheduling applications available in CPLEX CP Optimizer. Scheduling models in CPLEX CP Optimizer follow the same basic paradigm as classical Constraint Programming. Specifically, models consist mainly of decision variables or decision expressions (in green in Figure 1) and constraints (in dark blue). The other modeling concepts that do not fall into these categories are grouped as "data structures" (in light blue). These data structures are matrices (transition distance matrices) or scalar functions (stepwise or piecewise linear functions) used to parameterize variables, expressions, or constraints in a model. The lines that connect the various features represent relationships between features. For example, sequence variables can be used in sequencing constraints and no-overlap constraints, and a no-overlap constraint can incorporate a transition distance matrix.

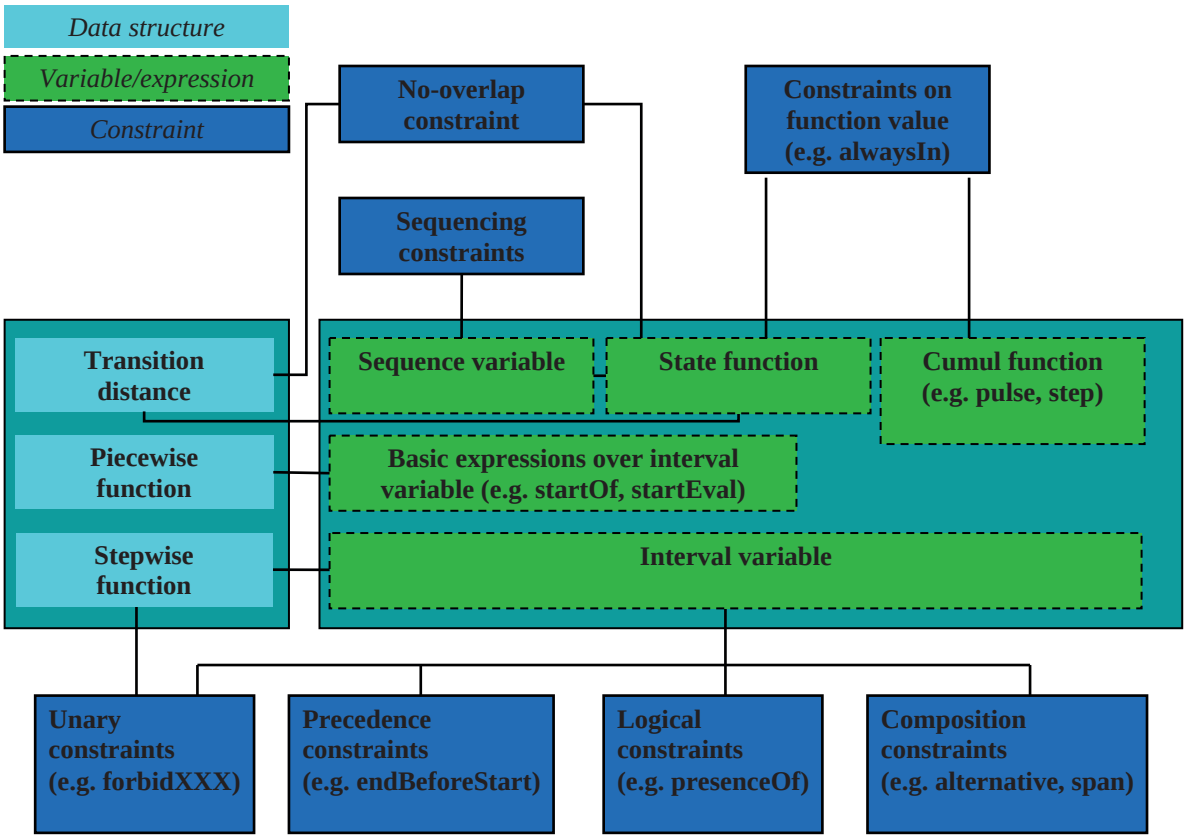


Figure 1: Overview of IBM ILOG CPLEX CP Optimizer modeling features for scheduling

Each of the examples to follow refers back to, and elaborates on, the relevant features from Figure 1.

Note:

- The feature or keyword discussions accompanying each example are relevant to the particular example, and typically only represent a subset of a keyword's functionality. Please refer to the documentation available in IBM ILOG CPLEX Optimization Studio for a complete description of each feature.
- While working on a model in OPL, once the dynamic help window is open (**Help** > **Dynamic Help**), you can access contextual help on any keyword by simply highlighting the word and clicking F1 twice.

Tables 1 and 2 provide a reference to the language features and modeling patterns used in each example.

Table 1: Mapping of language features to examples

	Oversubscribed Scheduling	Project scheduling	Transportation scheduling	Personal task scheduling	Batch scheduling
Functions					
piecewise linear function					
stepwise function		X		X	
Interval variable					
range	X		X	X	
size	X	X	X	X	X
intensity		X			
optional status	X	X	X	X	X
Constraints on interval variables					
precedence		X	X	X	X
forbidStart/End/Extent				X	
presenceOf	X	X	X	X	X
span		X	X	X	
alternative	X	X	X		X
synchronize					
Expressions on interval variables					
start/end/...Of		X	X	X	X
start/end/...Eval					
Sequence variable					
basic				X	
transition types				X	
Constraints on sequence variables					
first/last/previous				X	
basic noOverlap			X		
noOverlap w/ transition distance				X	

	Oversubscribed Scheduling	Project scheduling	Transportation scheduling	Personal task scheduling	Batch scheduling
Elementary cumul functions					
variable cumul functions					
constant cumul functions	X				X
Expressions on cumul functions					
heigthAtStart/End					
Cumul function expression					
cumul	X				X
Constraints on cumul expressions					
global: <=, >=	X				
alwaysIn					
State functions					
alwaysEqual					X
alwaysConstant					
alwaysIn					
alwaysNoState					
alignments					
transition distance					X

Table 2: Mapping of modeling patterns to examples

	Oversubscribed Scheduling	Project scheduling	Transportation scheduling	Personal task scheduling	Batch scheduling
Resource calendars		X		X	
Work breakdown structures		X			
Multi-mode		X			
Vertical alternatives	X	X			X
Horizontal alternatives			X		
Optional chains			X	X	
Batches					X
Resource allocation costs		X			
Activity execution costs	X		X		

Example 1: Oversubscribed scheduling

In this type of scheduling problem, the cumulative demand for resources exceeds their availability and the objective is to schedule as many as possible tasks given the constrained resources. This paper demonstrates modeling this type of problem using the United States Air Force (USAF) Satellite Control Network scheduling problem (Kramer *et al.*, 2007).

Demonstrated features

The light bulbs in Figure 1.1 highlight the main CPLEX CP Optimizer features used to model this problem. The features that are not yet introduced are grayed out.

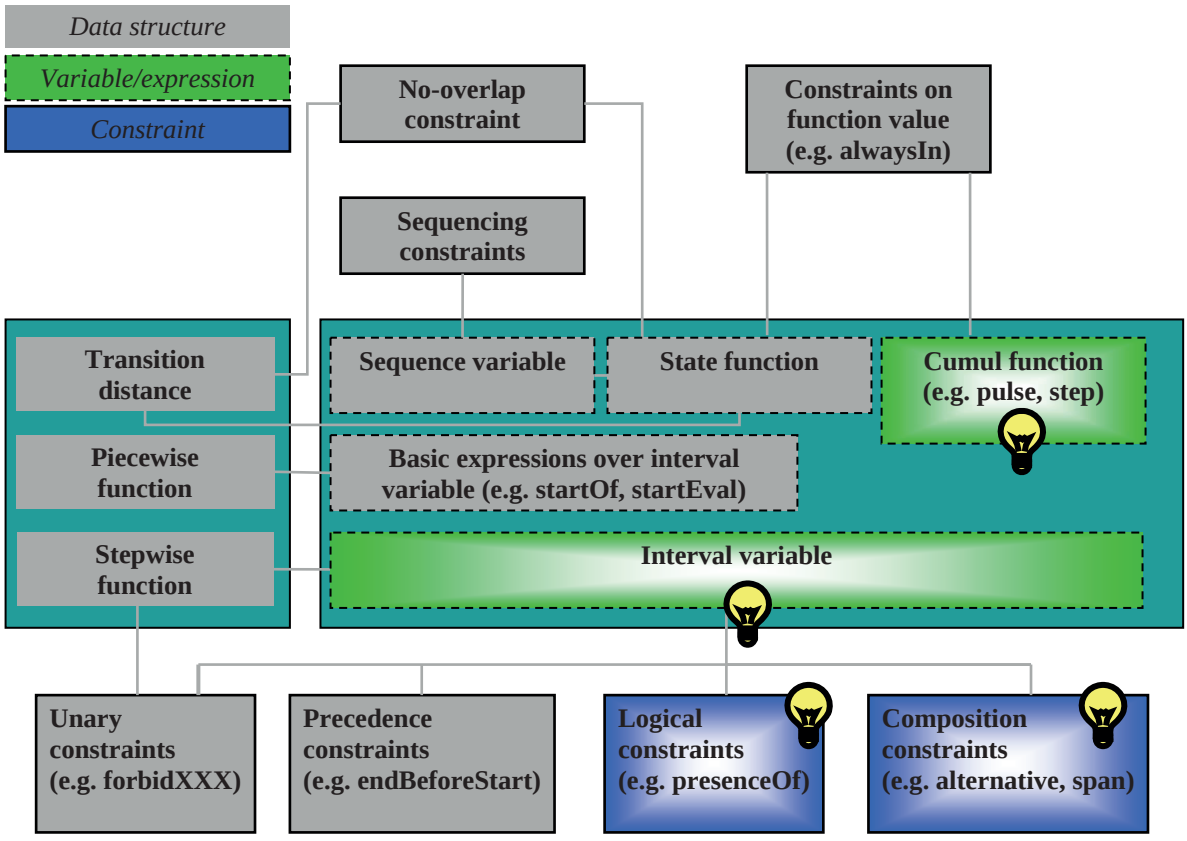


Figure 1.1: Modeling features demonstrated in the oversubscribed scheduling example.

The business problem

A set of communication requests for Earth-orbiting satellites must be scheduled on a total of 32 antennas spread across 13 ground-based tracking stations. At any point in time, it is possible that the number of requests exceeds the number of available antennas. The objective is to maximize the number of satisfied requests over the scheduling horizon. The problem can be specified as follows:

- Each station has a number of antennas that can be used to fulfill requests.
- Each communication request requires one antenna from the station it is scheduled on.
- To fulfill a specific request, only one of a given set of alternatives can be used. Each alternative specifies:
  - A station that can be used for that request
  - A time window during which the request must be fulfilled if that alternative is chosen
  - A duration
- All requests are optional.

The data

Tables 1.1 and 1.2 show representative data for this problem. The corresponding data element names as used in the OPL projects are listed in brackets under the column name. Table 1.1 contains data for the stations, while Table 1.2 contains a list of alternatives for completing communication requests. Each alternative represents a possible assignment of a station, time interval, and duration for completing the request. For example, communication request 239 can be performed by completing at most one out of two alternatives, namely either on station 6 or on station 9, both with earliest start 13140, duration 4097, and latest end 19440. The complete data set can be viewed in the OPL .dat files.

Table 1.1: Station data

Station name (`name`)	Station id (`id`)	Number of antennas (`capacity`)
POGO	1	13
GUAM	7	5
HAWK	10	20

Table 1.2: Communication request data

Request id (`task`)	Station id (`id`)	Earliest possible start time (`smin`)	Request duration (`duration`)	Latest possible end time (`emax`)
1	6	1800	1422	3600
239	6	13140	4097	19440
239	9	13140	4097	19440
239A	6	16200	4162	22500
456	2	87120	1077	89820

What is the objective?

The objective is to maximize the number of completed communication requests. In other words, maximize the number of requests included in the schedule.

What are the decisions to be made?

The decisions to include in the model are:

- Which requests to include in the schedule
- Which alternative (the combination of task, station, time interval and duration) to choose for each scheduled request
- The exact start times for the scheduled alternatives

What are the constraints?

The constraints specify that:

- Each scheduled request must be scheduled using one of its specified alternatives.
- The number of requests scheduled on a station cannot exceed the capacity (number of antennas) of that station at any point in time.

OPL files

The OPL projects to use with this example are:

- <*installDir*>`/Steps/SatelliteScheduling/Satellite scheduling STEP 1:` This OPL project is the starting point for completing the model. It includes all the data declarations and preprocessing, as well as comments to help you complete the objective, decision variable and constraint declarations.
- <*installDir*>`/Solutions/SatelliteScheduling/Satellite scheduling SOL 1:` The final solution.

<*installDir*> is the directory where you downloaded the attached files to.

Take a look at the Satellite scheduling STEP 1 project in ICPLEX Optimization Studio by doing the following:

1. Launch IBM ILOG CPLEX Optimization Studio.

2. Import the Satellite scheduling STEP 1 project by choosing **File** > **Import** > **Existing OPL projects**, selecting the <*installDir*>`/Steps/SatelliteScheduling` directory, and choosing the STEP 1 Satellite scheduling project.

3. Take a look at both the model (.mod) and data (.dat) files to get a feel for the data representation and steps required to complete the model.

There are two run configurations: the default run configuration, as well as a run configuration that uses a reduced data set that can be used if you are working with a trial version of the software.

Note: In the model, the word **`task`** is used to represent communication requests. For the remainder of this example, wherever the word “task” is used, it can be understood to refer to a communication request.

Model the decision variables using `interval`

The `interval` keyword

An **`interval`** variable represents an interval of time during which, for example, a task occurs. The general OPL syntax for declaring an **`interval`** variable **`a`** is as follows:

```
dvar interval a   [optional[(IsOptional)]] // For optional intervals
                  [in StartMin..EndMax] // Earliest start and latest end
                  [size SZ | in SZMin .. SZMax] // Duration
                  [intensity F]; // Intensity
```

Note: The **`optional`** attribute is especially useful for over-subscribed scheduling problems where not all the tasks can be performed, i.e. where some or all of the tasks must be optional.

For more information on the **`interval`** variable, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL keywords > **interval**

Steps to complete the OPL model

In the satellite scheduling example, each task is associated with a number of alternatives for completing the task. In addition to an **`interval`** variable for each task, an **`interval`** variable is needed for each alternative. Complete the following steps in the OPL model:

1. Declare **`interval`** variables to represent each task.

2. Declare **`interval`** variables to represent each alternative.

Be sure to capture the following in the declarations:

- All tasks and alternatives are optional.
- Each scheduled alternative should occur after its earliest start time.
- Each scheduled alternative should occur before its latest end time.
- Each scheduled alternative has a given duration.

The solution

The decision variables for this problem can be modeled using two arrays of **`interval`** variables. The first array represents the tasks and this declaration includes the keyword **`optional`**, because the problem is oversubscribed and all the tasks are therefore optional:

```
dvar interval task[t in Tasks] optional;
```

The second array represents the alternatives for completing the tasks. These variables are associated with specific ground stations, as can be seen in the set of **`Alternatives`**, and will be used later to write the capacity constraints of the ground stations. The **`alts`** variables are also optional, because at most one alternative can be chosen for each task. This declaration

concisely expresses the earliest start times **(smin)** and latest end times **(emax)** using the **in** keyword, as well as the durations **(duration)** using the size keyword:

```
dvar interval alt[a in Alternatives]
optional in a.smin..a.emax size a.duration;
```

Note that the intensity is not relevant for this particular problem.

Model the objective using presenceOf

The objective is to maximize the number of communication requests fulfilled, in other words the number of tasks included in the schedule.

The presenceOf keyword

CPLEX CP Optimizer provides a keyword to determine whether an **interval** variable is present in a solution or not, namely **presenceOf**. The constraint **presenceOf(a)** returns true if interval **a** is present in the schedule, and false otherwise.

For more information on **presenceOf**, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **presenceOf**

Steps to complete the OPL model

Use the **presenceOf** keyword to formulate the objective.

The solution

The objective maximizes the total number of tasks included in the schedule:

```
maximize sum(t in Tasks)
presenceOf(task[t]);
```

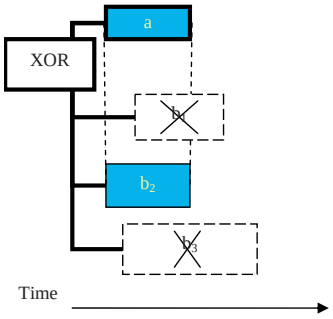
Model the constraints

Use alternative to model the relationship between the task intervals and the alt intervals

The first constraint states that each scheduled request must use one of its alternatives, in other words an exclusive or relationship between all **alt** intervals that apply to the same **task**.

The alternative constraint

The constraint **alternative(a, {b1, .., bn})** models an exclusive or between **{b1, .., bn}**. If interval **a** is present then exactly one of intervals **{b1, .., bn}** is present and **a** starts and ends together with the chosen one. If **a** is not present, then no **b** interval is present. This is shown in Figure 1.2. This is a vertical alternative, which is typically used when the alternative represents different resource modes or recipes for executing an activity. Later in this paper (see the *Transportation scheduling* example) you will see an example of a horizontal alternative, which is typically used to represent a discrete set of positions of a task or activity in time.



Example:
alternative(a,[b_1,b_2,b_3]);

a and **b_2** are executed simultaneously. No other intervals are executed.

Figure 1.2: The alternative constraint

For more information on the **alternative** constraint, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **alternative**

Steps to complete the OPL model

Use the **alternative** constraint to express the relationship between the **interval** variables representing requests **(task)** and the **interval** variables representing alternatives **(alt)** for completing each task. **Hint:** Use the **all** keyword to select only the alternatives in the **Alternatives** array corresponding to the relevant task.

The solution

The **alternative** constraint is used to model the exclusive or relationship that states that each scheduled task must use one of its alternatives. The **all** keyword is used to select only the alternatives in the **Alternatives** array corresponding to the relevant task:

```
forall(t in Tasks)
      alternative(task[t], all(a in
      Alternatives: a.task==t) alt[a]);
```

Use pulse to model cumulative capacity usage of the ground stations

The second constraint states that the number of requests scheduled on a station cannot exceed the station capacity at any point in time. Because the station data is associated with each alternative as opposed to each request, and because each scheduled request will correspond to at most one alternative, this constraint can be rephrased to state that the number of alternatives scheduled on a station cannot exceed the station capacity at any point in time.

The pulse function

The contribution of any one interval to the capacity usage can be modeled using a **pulse** expression. **pulse** is an elementary cumulative function expression representing the contribution of an individual **interval** variable or fixed interval of time. For example, the expression **pulse(a, h)** specifies that an interval **a** contributes a pulse function of height **h**, as shown in Figure 1.3.

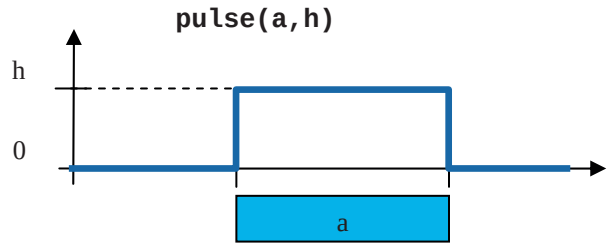


Figure 1.3: An example of the pulse expression

For more information on **pulse**, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **pulse**

Steps to complete the OPL model

1. Use the **pulse** expression to model the contribution of each alternative to each station's capacity usage.

2. Specify that for each station, the sum of such contributions from all alternatives associated with that station should be less than or equal to the station's capacity.

The solution

For the capacity constraint, the **pulse** cumulative function is used to model a contribution of 1 for each alternative, because each alternative will use 1 antenna. Next, the contributions from all alternatives scheduled on a given station are summed up and required to be less than or equal to the station capacity. The **optional** attribute in the **alt** variable declaration implies that contributions will only be added for alternatives actually present in the schedule:

```
forall(s in Stations)
      sum(a in Alternatives:
      a.station==s.id) pulse(alt[a],1)
      <= s.capacity;
```

The final OPL model

```
using CP;

/******/
* Data *
*******/

tuple GroundStation {
 string name; // name of the station
 key int id; // station identifier
 int capacity; // number of antennas
};

tuple Alternative {
 string task; // task identifier
 int station; // station identifier
 int smin; // earliest possible start time
 int duration; // task duration
 int emax; // latest possible end time
};
```

Run the model

If you have not already done so, complete the model file of the Satellite scheduling STEP 1 project and run the model. If you are having any problems, import and look at the solution in the Satellite scheduling SOL 1 project, available in the Solutions directory, to fix any errors in your model.

Look at the **Scripting log** and see that nearly all the tasks (1353 out of 1371) have been scheduled within the 2-minute time limit (this limit is set in the .ops file at **Constraint Programming** > **Search Control** > **Limits** >> **Time limit**). In the **Problem browser**, scroll down to the decision variables and hover the mouse on the variable name to see the icon for displaying the data view. For each variable, click this icon to view the solution, namely the chosen tasks and alternative for completing each task.

```
{ GroundStation } Stations = ...; // list of stations
{ Alternative } Alternatives = ...; // list of task alternatives

{ string } Tasks = { a.task | a in Alternatives }; // list of tasks
int NbTasks = card(Tasks); // number of tasks

/*********************
* Decision variables *
*********************/

dvar interval task[t in Tasks] optional;
dvar interval alt[a in Alternatives] optional in a.smin..a.emax size a.duration;

/*********************
* Objective function *
*********************/

maximize sum(t in Tasks) presenceOf(task[t]);

/**************
* Constraints *
***************/

subject to {

 forall(t in Tasks)
  alternative(task[t], all(a in Alternatives: a.task==t) alt[a]);

 forall(s in Stations)
    sum(a in Alternatives: a.station==s.id) pulse(alt[a],1) <=
    s.capacity;

};
```

Summary

This example demonstrated how to use CPLEX CP Optimizer to efficiently model and solve an oversubscribed satellite scheduling problem. The modeling patterns that were demonstrated in this example include:

- Exclusive or
- Cumulative capacity usage

Example 2: Project scheduling

A project is defined in terms of a hierarchical set of tasks (the work breakdown structure), with precedence constraints between tasks. The goal is to assign workers to tasks so as to minimize both the project makespan (the total time to complete the project) and the total worker cost, with some constraints involving the workers' proficiencies. You will see how to build a basic project scheduling model and learn how to extend the model to incorporate constraints on workers' schedules. You will also learn how to extend the model to incorporate the ability to cancel certain subprojects at a fixed cost.

Demonstrated features

The light bulbs in Figure 2.1 highlight the main CPLEX CP Optimizer features used to model this problem. The features that are not covered in this or the preceding example are grayed out.

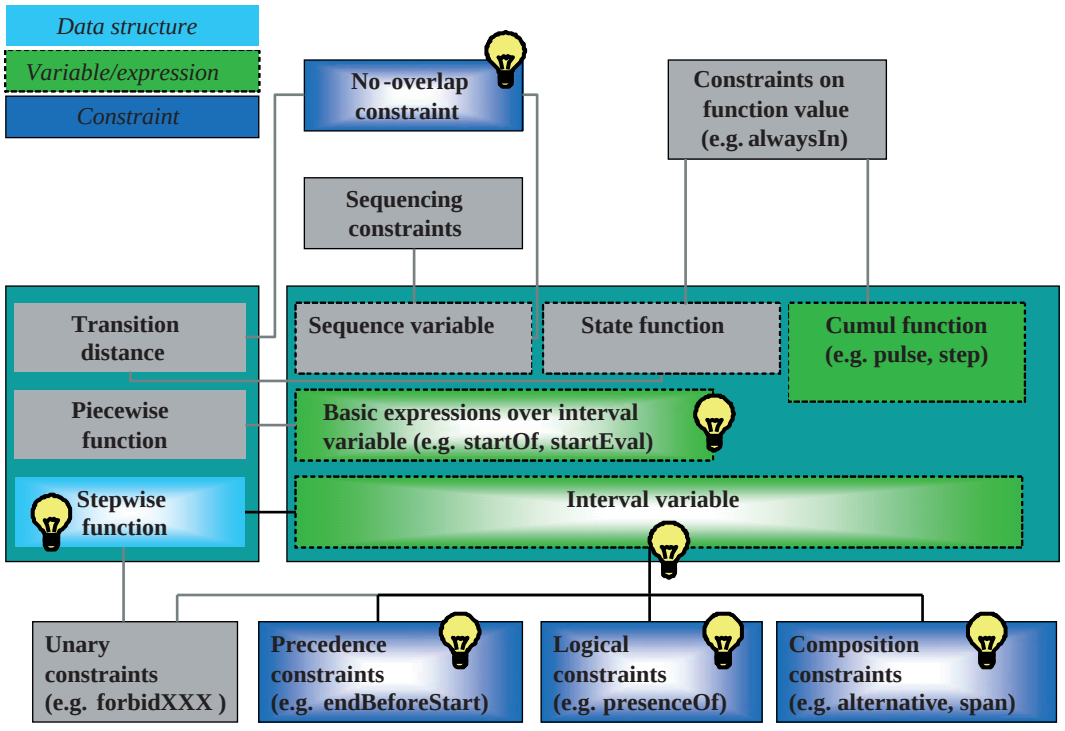


Figure 2.1: Modeling features demonstrated in the project scheduling example

The business problem

Your client needs a tool to schedule a set of subprojects, each consisting of a set of hierarchical tasks, with precedence constraints between tasks and between tasks and subprojects. Such a work breakdown structure is shown in Figure 2.2, where the dashed lines represent precedence constraints. Note that in this example the precedence constraints can cross several levels.

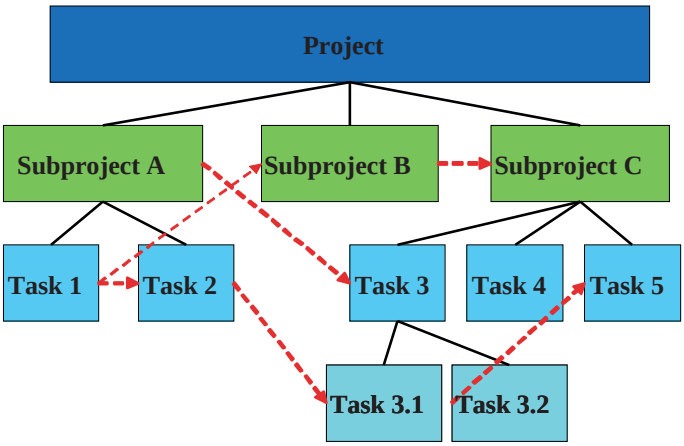


Figure 2.2: Work breakdown structure for project scheduling

The scheduling problem has the following additional characteristics:

- Workers have to be assigned to perform the tasks.
- Each worker has a set of skills, and a certain proficiency level for each skill.
- A holiday calendar specifies breaks during which workers are not available.
- Each task requires one or more workers with specified minimum or maximum proficiency level, or both, for some skills.
- Certain subprojects are not considered mandatory and may be excluded at an additional fixed cost.
- Hiring workers involves two types of cost:
  - A fixed cost that must be paid if a worker is hired for the project.
  - A proportional cost that depends on the total duration of the tasks performed by the worker.
- The objective is to minimize a weighted sum of:
  - The project makespan.
  - The total worker cost (fixed and proportional).
  - The cost of not executing the non-mandatory subprojects.

To demonstrate how to model this problem, it is broken into the following smaller, more manageable steps:

- Step 1: Work breakdown structure and minimizing the makespan
- Step 2: Worker assignments and costs
- Step 3: Worker calendars
- Step 4: Non-mandatory subprojects and non-execution costs

Each of these steps includes descriptions of the data required to complete that part of the model, as well as the relevant definitions of the objective, decision variables, and constraints.

Step 1: Work breakdown structure and minimizing the makespan

The focus in this step is only on modeling the work breakdown structure, together with the precedence constraints. The assignment of workers to tasks is not yet addressed. This step includes the objective of minimizing the makespan.

The data

Tables 2.1 through 2.4 show representative data required to model the work breakdown structure, the precedence constraints, and the objective of minimizing the makespan. The corresponding data element names as used in the OPL projects are listed in brackets under the column names. The complete data set used for this example can be found in the ProjectData.xls file available in the `<installDir>/Steps/ProjectScheduling/data directory`, where `<installDir>` is the directory where you downloaded the attached files to.

Table 2.1: Task data

Task id (`id`)	Task name (`name`)	Minimum duration (hrs) (`ptMin`)
4	Task 4	248
5	Task 5	180
6	Task 6	147

Table 2.2: Hierarchy data

Task id (`taskId`)	Parent task id (`parentId`)
4	3
3	2

Table 2.3: Task precedence data

Task 1 id (`beforeId`)	Task 2 id (`afterId`)	Type of precedence constraint (`type`)	Delay between tasks (hrs) (`delay`)
4	5	StartsAfterEnd	0
95	96	StartsAfterStart	6

Table 2.4: Objective weights

KPI	Weight
Makespan	1

What is the objective?

The objective is to minimize the makespan, that is, the end time of the last task in the schedule.

What are the decisions to be made?

At this point, the only decisions that need to be made are the start and end times of the tasks.

What are the constraints?

The constraints required at this point are:

- Constraints describing the work breakdown structure
- The precedence constraints

OPL files

Use the following OPL projects:

- `<installDir>/Steps/ProjectScheduling/STEP1 WorkBreakdownStructure/Project scheduling STEP 1`: This OPL project is the starting point for completing the model. It includes all the data declarations and preprocessing required for this step, as well as comments to help you complete the objective, decision variable and constraint declarations.
- `<installDir>/Solutions/ProjectScheduling/STEP1 WorkBreakdownStructure/Project scheduling SOL 1`: The solution to Step 1.

Take a look at the Project scheduling STEP 1 project in CPLEX Optimization Studio by doing the following:

1. Launch CPLEX Optimization Studio.

2. Import the Project scheduling STEP 1 project by choosing **File** > **Import** > **Existing OPL projects**, selecting the `<installDir>/Steps/ProjectScheduling/STEP1 WorkBreakdownStructure directory`, and choosing the `Project scheduling STEP 1` project.

3. Take a look at both the model (.mod) and data (.dat) files to get a feel for the data representation and steps required to complete the model.

Each step in this example contains two run configurations: a default run configuration, as well as a run configuration that uses a reduced data set that can be used if you are working with a trial version of the software.

Model the decision variables

For the project scheduling problem, an **`interval`** variable can be used for the time interval during which each task occurs. The relevant OPL syntax was covered in the *Oversubscribed Scheduling* example, and more information can be found in the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL keywords > **interval**

Steps to complete the OPL model

Declare **interval** variables to represent each task. Be sure to capture the minimum duration in the declaration.

The solution

The decision variables for this problem can be modeled as an array of **`interval`** variables, indexed over the set of **`Tasks`**. The **`size`** keyword is used to express the minimum task duration (**`t.ptMin`**). A large value (**`MaxInterval`**) is used as the upper bound on the duration of a task, because tasks may be split into several parts, thus increasing the duration to an unknown value.

```
dvar interval task[t in Tasks] size
t.ptMin..MaxInterval;
```

Model the objective using `endOf`

The `endOf` keyword

The objective is to minimize the makespan, in other words, to minimize the end of the task completed last in the schedule. CPLEX CP Optimizer provides the **`endOf`** keyword that can be used for this purpose. The expression **`endOf(a)`** returns an integer value equal to the end of the interval **`a`**.

For more information on endOf, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL keywords > **endOf**

Steps to complete the OPL model

1. Write a decision expression that uses the **`endOf`** keyword to define the makespan.

2. Use the **`makespan`** decision expression, together with the **`MakespanWeight`** (as already declared in the model) to define the objective function.

The solution

The following decision expression defines the makespan as the maximum end time among all the tasks:

```
dexpr int makespan = max(t in Tasks)
endOf(task[t]);
```

The objective minimizes the makespan, multiplied by its weight. While the weight does not influence the optimization at this point, it will be used in the next step to express the tradeoff between the makespan and the total worker cost.

```
minimize MakespanWeight * makespan;
```

Model the constraints

Model the work breakdown structure using `span`

The work breakdown structure for this project scheduling problem is defined by a hierarchical set of tasks: At the top of the hierarchy, the top-level task (in this case the project) spans

the subprojects. In turn, each subproject spans the tasks it is made up of, and so forth. CPLEX CP Optimizer provides the **span** keyword that can be used to concisely model such a structure.

The span constraint

The **span** keyword is used to express the constraint that one interval must span the start and end of a group of other intervals. The constraint **span(a, {b1,…,bn})** means that if interval **a** is present, it must span all present intervals from the group **{b1,…,bn}**. At least one interval $\mathbf{b_i}$ coincides with the start of **a**, and at least one interval $\mathbf{b_i}$ coincides with the end of **a**. If **a** is absent, all intervals $\mathbf{b_i}$ are absent. If **a** is present, at least one of the intervals $\mathbf{b_i}$ is constrained to be present. An example of the **span** constraint is shown in Figure 2.3.

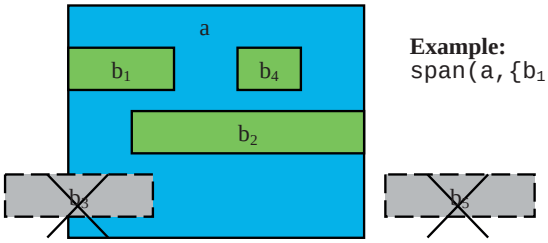


Figure 2.3: An example of the span constraint

For more information on the **span** constraint, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **span**

Steps to complete the OPL model

For each parent task, *p*, use the **span** constraint to specify that this task must span the group of tasks for which *p* is a parent. **Hint:** Use the **all** keyword to select only the tasks that have task *p* as a parent.

The solution

The constraint is written for all parent tasks, and specifies that the task spans all its children.

```
forall (t in Tasks : t.id in Parents)
   span(task[t],
       all(i in ParentLinks: i.parentId ==
       t.id) task[<i.taskId>]);
```

Model the precedence constraints

Precedence constraints available in IBM ILOG CPLEX CP Optimizer

CPLEX CP Optimizer provides several precedence constraints to model the position of one task relative to another, as summarized in Figure 2.4. For example, the precedence constraint **startBeforeStart(a,b,10)** states that the start of interval a must occur at least 10 time units before the start of interval **b**.

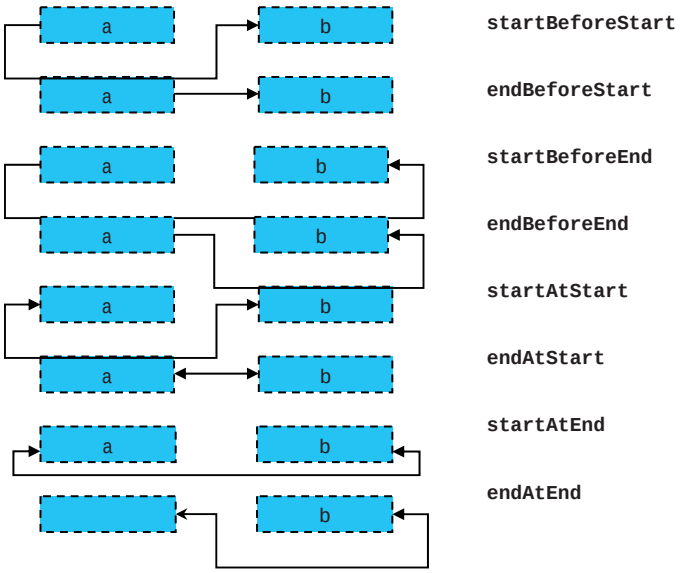


Figure 2.4: IBM ILOG CPLEX CP Optimizer precedence constraints

For more information on precedence constraints, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Reference Manual > OPL, the modeling language > Scheduling **Precedence constraints between interval variables**

Steps to complete the OPL model

1. Look at the precedence data in the Excel spreadsheet and determine which types of precedence constraints match the descriptions in the data. Note that the descriptions in the data do not necessarily match the OPL keywords used to express precedence constraints.

2. For each type of precedence, write the relevant precedence constraint for all elements from the set **`Precedences`** for which the type equals the description of that **`type`** of precedence in the data.

The solution

There are only two types of precedence given in the data, namely "StartsAfterStart" and "StartsAfterEnd." Neither of these descriptions corresponds directly to the keywords used to express precedence constraints, but the **`beforeID`** column indicates which task comes first and one can therefore find the corresponding OPL constraint. The first type of precedence states that the start of **`afterID`** must start after the start of **`beforeID`**. This is equivalent to stating that the start of **`beforeID`** must start before the start of **`afterID`**, and can be expressed using the **`startBeforeStart`** constraint:

```
forall (p in Precedences : p.type == "StartsAfterStart")
    startBeforeStart(task[<p.beforeId>], task[<p.afterId>], p.delay);
```

Similarly, the data with description "StartsAfterEnd" can be modeled using the **`endBeforeStart`** constraint:

```
forall (p in Precedences : p.type == "StartsAfterEnd")
    endBeforeStart(task[<p.beforeId>], task[<p.afterId>], p.delay);
```

Run Step 1

If you have not already done so, complete the model in the `Project scheduling STEP 1` model file and run the model. If you are having any problems, import and look at the solution in the `Project scheduling SOL 1` project, available in the `Solutions` directory, to fix any errors in your model.

In the **Problem browser**, scroll down to the decision variables and hover the mouse on the **`task`** variable name to see the icon for displaying the data view. Click this icon to view the solution. Notice that the top-level task (task id 1) has an end time of 22919. In the next step, you will add the worker assignments to the model and see whether the makespan changes.

Step 2: Worker assignments and costs

You will complete the basic project scheduling model by adding the worker assignments and costs.

The data

Tables 2.5 through 2.10 show representative data related to the worker assignments. The corresponding data element names as used in the OPL projects are listed in brackets under the column names.

Table 2.5: Worker data

Worker id (`id`)	Worker name (`name`)	Fixed cost (`fixed Cost`)	Proportional cost (`propCost`)
1	Worker 1	31000	7

Table 2.6: Skills data

Skill id (`id`)	Skill name (`name`)
1	Skill 1

Table 2.7: Worker skill proficiency data

Worker id (`workerId`)	Skill id (`skillId`)	Proficiency level (`level`)
3	2	2

Table 2.8: Requirement – task mapping

Requirement id(`id`)	Task id (`taskId`)
9	12
10	12

Table 2.9: Requirements defined in terms of skills and required proficiencies

Requirement id (`reqId`)	Skill id (`skillId`)	Minimum proficiency level (`levelMin`)	Maximum proficiency level (`levelMax`)
9	7	0	1000
71	3	2	5

Table 2.10: Objective weights

KPI	Weight
Makespan	1
WorkersFixed	0.04
WorkersProportional	0.01

What is the objective?
The objective is to minimize the weighted sum of the makespan and the total worker cost, where the total cost consists of:

- A fixed cost that must be paid if a worker is hired for the project.
- A proportional cost that depends on the total duration of the tasks performed by the worker.

What are the decisions to be made?
The decisions to be made, in addition to those from Step 1, are:

- Which workers to use for the project
- Which worker to assign to each task

What are the constraints?
The additional constraints involving the worker assignments are:

- Each worker can work on at most one task at any point in time.
- Each skill requirement for each task must be met by one of the workers with corresponding proficiency for that skill.

OPL files
Use the following OPL projects:

- `<installdir>/Steps/ProjectScheduling/STEP2 Workers/Project scheduling STEP 2`: This OPL project contains all the code from Step 1 and is the starting point for completing Step 2. It includes all the additional data declarations and preprocessing required for this step, as well as comments to help you complete the model.
- `<installdir>/Solutions/ProjectScheduling/STEP2 Workers/Project scheduling SOL 2`: The solution to Step 2.

Import the `Project scheduling STEP 2` project in CPLEX Optimization Studio from the `<installdir>/Steps/ProjectScheduling/STEP2 Workers` directory. Take a look at both the model (.mod) and data (.dat) files to get a feel for how the worker-related data is handled. Note especially the new sets, **`PossibleWorkers`** and **`Allocations`**, that are used to create all possible assignments of workers to task requirements.

Model the decision variables

Use `interval` variables

In order to determine the fixed worker cost, you have to know whether each worker is used for the project or not. This can be modeled using an optional **`interval`** variable to represent the total period of time during which each worker is used in the project. A constraint, to be modeled later in this step, will specify that each such interval variable must span all the tasks the worker is assigned to.

In addition, **`interval`** variables can be used to represent the time intervals during which workers are allocated to task requirements. Because several alternative worker assignments are possible for a given requirement, these interval variables must be declared as optional: only the intervals representing a selected worker for a requirement will be present in the final solution.

The OPL syntax for declaring an interval variable is given earlier in this example, and more information can be found in the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL keywords > **interval**

Steps to complete the OPL model

1. Declare optional **`interval`** variables to determine whether each worker is used in the project.

2. Declare optional **`interval`** variables to represent the time intervals during which workers are allocated to task requirements. **Hint:** The set **`Allocations`** contains all possible allocations of workers to task requirements.

The solution

The array of optional **`interval`** variables, **`workerSpan`**, is indexed over the set of **`Workers`** and represents the time intervals to determine whether each worker is used in the project:

```
dvar interval workerSpan[w in Workers]
optional;
```

Another array of optional **`interval`** variables, **`alts`**, indexed over the set of **`Allocations`**, is used to determine the allocation of workers to task requirements. Each array element represents a possible allocation of a worker to a task requirement.

```
dvar interval alts[a in Allocations]
optional;
```

Model the objective

The new objective must include additional terms to minimize both the fixed and proportional worker costs. In order to apply the fixed cost, the **`presenceOf`** constraint can be used to determine whether each **`workerSpan`** variable is present or not. The **`presenceOf`** constraint is discussed in the Oversubscribed Scheduling example, and you can also find more information in the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **presenceOf**

The variable cost calculation requires the total time spent by each worker. This can be calculated using the **`sizeOf`** keyword.

The `sizeOf` keyword

The expression **`sizeOf(a, 0)`** returns the size of interval **`a`** if it is present in the schedule. If **`a`** is absent, the specified default value (in this case 0) will be used. It is possible to specify a default other than 0.

For more information on the **`sizeOf`** keyword, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > **OPL** functions > **sizeOf**

Steps to complete the OPL model

1. Write a decision expression to calculate the **`workersFixedCost`**. **Hint:** Use the **`workerSpan`** variables and **`presenceOf`** constraint.

2. Write a decision expression that uses the **`sizeOf`** keyword to calculate, for each worker, the total time spent on task requirements (**`workerTimeSpent`**).

3. Write a decision expression that calculates the total proportional cost (**`workersPropCost`**) to be the sum over all workers, of the proportional cost coefficient (**`propCost`**) multiplied by the total time worked (**`workerTimeSpent`**).

4. Complete the objective function by adding the **`workersFixedCost`** and **`workersPropCost`** multiplied by their respective weights.

The solution

The following three decision expressions are used to calculate the **workersFixedCost** and **workersPropCost**:

```
dexpr float workersFixedCost =
    sum(w in Workers) w.fixedCost * presenceOf(workerSpan[w]);

dexpr int workerTimeSpent[w in Workers] =
    sum(a in Allocations: a.workerId==w.id) sizeOf(alts[a],0);

dexpr float workersPropCost =
    sum(w in Workers) w.propCost*workerTimeSpent[w];
```

These are then added to the objective, multiplied by their respective weights:

```
minimize MakespanWeight * makespan
         + FixedWeight * workersFixedCost
         + ProportionalWeight * workersPropCost;
```

Model the constraints

Use span to determine whether each worker is used

In order to model the fixed cost, you declared an **interval** variable called **workerSpan**. This variable must be present in the schedule if the worker has been allocated to any tasks. The **span** constraint, discussed earlier in this example, can be used to enforce this. Specifically, if some tasks are present for this worker, the **workerSpan** interval must span all the tasks executed by the worker (that is, it must start at the start time of the first task allocated to the worker and end at the end time of the last task).

Steps to complete the OPL model

For all workers, write a constraint that states that the **workerSpan** variable must span all **alts** variable for that worker.

The solution

In the constraint, the **all** keyword is used to select all the **alts** variables for the relevant worker:

```
forall(w in Workers)
  span(workerSpan[w], all(a in Allocations:
  a.workerId==w.id) alts[a]);
```

Use alternative to specify that each task requirement must be fulfilled by one worker

Each skill requirement for each task must be fulfilled by one of the workers with corresponding proficiency for that skill. The set of **Allocations** already captures all possible allocations of workers to task requirements, based on worker proficiencies. You therefore only need to specify that the task associated with a particular requirement must be completed using one of the allocations for that requirement. This is an "exclusive or" constraint that can be modeled using the **alternative** keyword.

A brief overview of **alternative** is given in the Oversubscribed Scheduling example, and more information can be found in the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **alternative**

Steps to complete the OPL model

For all **Requirements**, write an **alternative** constraint to specify that the **task** associated with a particular requirement must be fulfilled using one of the **Allocations** for the requirement.

The solution

The **all** keyword is used to select all **alts** variables for the particular requirement. The **alternative** constraint then specifies that the task associated with the requirement must be fulfilled using one of the selected **alts** variables:

```
forall(r in Requirements)
    alternative(task[<r.taskId>], all(a in
    Allocations: a.reqId==r.id) alts[a]);
```

Use noOverlap to model that a worker can work on only one task at a time

The **noOverlap** keyword can be used to model constraints on a unary resource, in this case the fact that each worker can work on only one task at a time.

The noOverlap constraint

If a set, **P**, of **interval** variables exists, then the expression **noOverlap(P)** implies that none of the **interval** variables in the set **P** may overlap, as shown in Figure 2.5.

Figure 2.5: An example of a solution for a noOverlap constraint

For more information on the noOverlap constraint, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > **OPL** functions > **noOverlap**

Steps to complete the OPL model

For all workers, write a constraint that states that none of the allocations (**alts** variables) involving the particular worker may overlap.

The solution

In the constraint, the **all** keyword is used to select all the **alts** variables for the particular worker:

```
forall(w in Workers)
    noOverlap(all(a in Allocations:
    a.workerId==w.id) alts[a]);
```

Run Step 2

If you have not already done so, complete the model in the `Project scheduling STEP 2` model file and run the model. If you are having any problems, import and look at the solution in the `Project scheduling SOL 2` project, available in the Solutions directory, to fix any errors in your model.

In the **Problem browser**, scroll down to the decision variables and hover the mouse on the **`task`** variable name to see the icon for displaying the data view. Click this icon to view the solution. Notice that the makespan has increased with the worker assignments in place. Next, look at the values of the **`workerSpan`** and **`alts`** variables to see which workers are working on the project, and fulfilling each task requirement. Look at the worker calendar data (which has not yet been incorporated into the model) and see in the solution for the **`alts`** variables whether you can find a worker that is scheduled to work during a break. Check whether the length (end time minus start time) of such an **`alts`** variable equals the sum of the interval size and the worker's break time. It is likely that there is a worker whose break time is violated, because calendars have not been modeled yet, and the solution to this step is likely invalid when the worker calendars are considered.

Note: Depending on the computer's processing speed, different solutions may be found. This is because a time limit of 1 minute is specified in the OPL settings (.ops) file. However, you should be able to observe inconsistencies between the worker assignments in this solution and the worker calendar data.

The next step demonstrates how to incorporate worker calendars into the model.

Step 3: Worker calendars

The problem remains mostly unchanged except that worker calendars, which specify when workers are taking breaks, are incorporated. In fact, the only thing that changes is the declaration of the **`alts`** variables, while the objective, constraints, and other variable declarations remain the same.

The data

Table 2.11 shows representative calendar data. The corresponding data element names as used in the OPL projects are listed in brackets under the column names.

Table 2.11: Calendar data

Worker id (`workerId`)	Break from (`start`)	Break to (`end`)
2	720	888

OPL files

Use the following OPL projects:

- `<installdir>/Steps/ProjectScheduling/STEP3 Calendars/Project scheduling STEP 3`: This OPL project is the starting point for this step and includes all the code from Step 2, as well as all the additional data declarations and preprocessing required.
- `<installdir>/Solutions/ProjectScheduling/STEP3 Calendars/Project scheduling SOL 3`: The solution to Step 3

Import the `Project scheduling STEP 3` project in CPLEX Optimization Studio from the *<installdir>*`/Steps/ProjectScheduling/STEP3 Calendars` directory. Take a look at both the model (.mod) and data (.dat) files to get a feel for the additional data representation and steps required to complete the model.

Model worker calendars using `stepwise` and `intensity`

With CPLEX CP Optimizer, calendars can be modeled simply and efficiently by using the **`intensity`** attribute of interval variables. A step function can be used to concisely express the calendar as a series of up and down steps, and this step function can then be used to define the intensity function of an interval variable.

The stepwise keyword

This keyword is used to express linear step functions. That is, piecewise linear functions where all the slopes are equal to zero. An example of the OPL syntax for declaring a stepwise expression is as follows:

```
stepFunction F = stepwise(i in 1..n)
{ V[i]->T[i]; V[n+1] };
```

Here, **`V[i]`** represents the height of step **`i`**, while **`T[i]`** represents the end of step **`i`**, as shown in Figure 2.6.

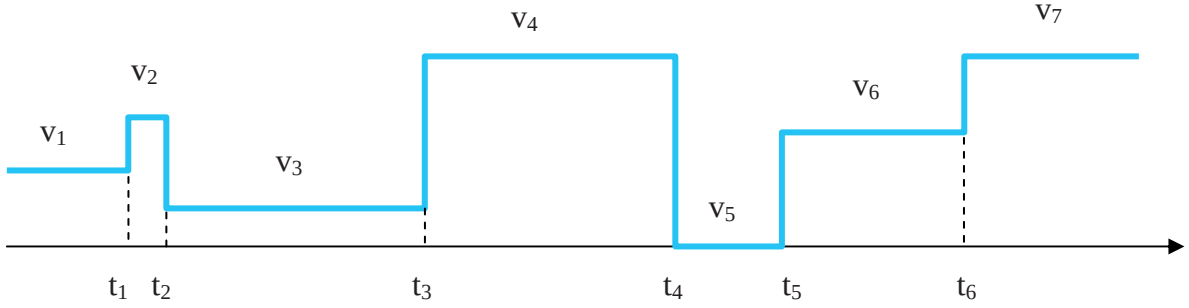


Figure 2.6: Example of a stepwise function

For more information on the stepwise keyword, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > **OPL** keywords > **stepwise**

The intensity keyword

The syntax for declaring an **`interval`** variable includes **`intensity`** as an optional attribute:

```
dvar interval a   [optional[(IsOptional)]] // For optional intervals
                  [in StartMin..EndMax] // Earliest start and latest end
                  [size SZ | in SZMin .. SZMax] // Duration
                  [intensity F]; // Intensity
```

Intensity is a function that applies a measure of usage or utility over an interval length. The intensity is 100% by default, and can not exceed this value. Consider an interval where a worker works full days during the week, and half days during the weekend. The intensity function would be 100% for five days and 50% for two days. In this example, the interval length is seven days and the size equals six days, as shown in Figure 2.7:

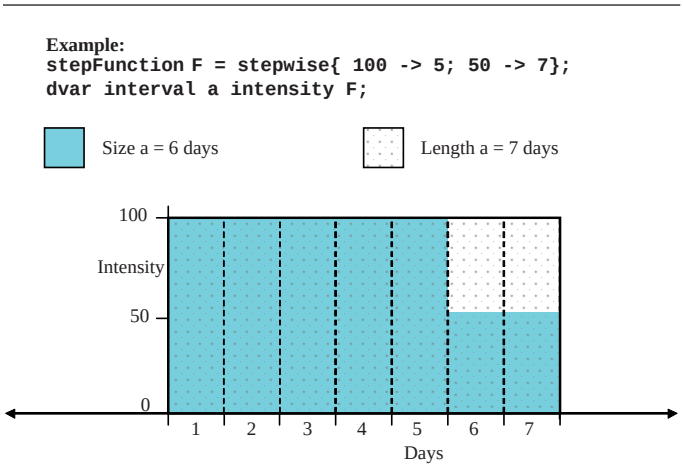


Figure 2.7: Example of applying `intensity` using `stepwise`.

For more information on the `intensity` keyword, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > **OPL** keywords > **intensity**

Steps to complete the OPL model

1. Using **`stepwise`** and the **`Steps`** array, write a **`stepFunction`** that describes the calendar for each worker.

2. Edit the declaration of the **`alts`** variables to model the intensity using the calendar step function.

The solution

Each element of the array **`Steps[w]`** is a sorted set of steps describing the calendar of the relevant worker, **`w`**. This array is used together with the **`stepwise`** keyword to model the calendar of each worker:

```
stepFunction calendar[w in Workers] =
stepwise (s in Steps[w])
{ s.v -> s.x; 100 };
```

Next, the **`calendar`** step function for a particular worker is used to define the **`intensity`** of the **`alts`** variables associated with that worker. The minimum duration of an **`alts`** variable (**`a.pt`**) is determined by a combination of the minimum processing time of the associated task and the individual calendars of workers for that allocation:

```
dvar interval alts[a in Allocations]
optional size a.pt..MaxInterval intensity
calendar[<a.workerId>];
```

Run Step 3

If you have not already done so, complete the model in the .mod file of the `Project scheduling STEP 3` project and run the model. If you are having any problems, import and look at the solution in the `Project scheduling SOL 3` project, available in the Solutions directory, to fix any errors in your model.

In the **Problem browser**, scroll down to the decision variables and hover the mouse on a variable name to see the icon for displaying the data view. Click this icon for each variable to view the solution. Notice that the inconsistencies between worker assignments and worker calendars are resolved. Any **`alts`** variable that overlaps a worker's break time will now have a length equal to sum of the interval size and the break time.

The next step demonstrates how to extend the model to incorporate the possibility of not executing certain subprojects at an additional cost.

Step 4: Non-mandatory subprojects and non-execution costs

In the first three steps of this example, all tasks were assumed to be mandatory. This final step involves modeling the fact that some tasks or subprojects are not mandatory and may be excluded from the schedule at a fixed cost. This problem can be addressed by declaring all tasks as **`optional`**, and then adding constraints forcing all mandatory tasks to be executed. A non-execution cost is then applied to all tasks not included in the schedule.

The data

Table 2.12 shows representative data of non-mandatory tasks (or subprojects) and the cost of not executing these tasks. Table 2.13 shows the weights to be used in the objective function. The corresponding data element as used in the OPL projects are listed in brackets under the column names.

Table 2.12: Non-mandatory tasks

Task id (`taskId`)	Non-execution cost (`nonExecCost`)
107	2681.1

Table 2.13: Objective weights

KPI	Weight
Makespan	1
WorkersFixed	0.04
WorkersProportional	0.01
NonExecution	1

What is the objective?

The objective is to minimize the weighted sum of the makespan, total worker costs, and task non-execution costs.

What are the decisions to be made?

The only additional decision to be made is which of the non-mandatory tasks not to execute.

What are the constraints?

The additional constraints are:

- The top-level task, representing the project as a whole, must be executed.
- Mandatory tasks must be executed if their parent is executed.

OPL files

Use the following OPL projects:

- `<installdir>/Steps/ProjectScheduling/STEP4 NonMandatoryTasks/Project scheduling STEP4`: This OPL project is the starting point for completing the model. It includes all the data declarations and preprocessing required for this step, as well as comments to help you complete the model.
- `<installdir>/Solutions/ProjectScheduling/STEP4 NonMandatoryTasks/Project scheduling SOL 4`: The solution to Step 4.

Import the Project scheduling STEP 4 project in CPLEX Optimization Studio from the `<installdir>/Steps/ProjectScheduling/STEP4 NonMandatoryTasks` directory. Take a look at both the model (.mod) and data (.dat) files to get a feel for the additional data representation and steps required to complete the model.

Model the decision variables

The only change required to the decision variables, is to declare the **task** variables as optional:

```
dvar interval task[t in Tasks] optional
size t.ptMin..MaxInterval;
```

Model the objective

The execution cost can be modeled by determining whether a non-mandatory task is present in the schedule or not.

Steps to complete the OPL model

1. Write a decision expression to calculate the non-execution cost based on whether a non-mandatory task is present in the schedule or not.

2. Add a term to the objective function to capture the weighted non-execution cost.

The solution

The **nonExecCost** decision expression calculates the non-execution cost for all tasks in the set of **NonMandatoryTasks**. The **presenceOf** keyword is used to test whether a task is part of the schedule or not: if the task is absent in the schedule, the non-execution cost is applied; otherwise, the cost is zero.

```
dexpr float nonExecCost = sum(n in
NonMandatoryTasks) n.nonExecCost*
(1-presenceOf(task[<n.taskId>]));
```

The objective then becomes:

```
minimize MakespanWeight * makespan
 + FixedWeight * workersFixedCost
 + ProportionalWeight * workersPropCos
 + NonExecutionWeight * nonExecCost;
```

Model the constraints

The constraints should have the effect that any mandatory tasks are included in the schedule if their parent is included.

Steps to complete the OPL model

1. Add a constraint stating that the top-level task must be present.

2. Add a constraint stating that any mandatory task must be present if its parent is present. **Hint:** Use the implication operator (=>).

The solution

The first constraint enforces the presence of the top-level task:

```
presenceOf(task[<TopLevelTask>]);
```

Next, any mandatory task is forced to be present if its parent is present. The combination of these two constraints forces the presence of mandatory tasks to trickle down from the top-level task, to its mandatory children, and so forth:

```
forall(p in ParentLinks: !p.taskId in
NonMandatoryTaskIds)
    presenceOf(task[<p.parentId>]) =>
    presenceOf(task[<p.taskId>]);
```

Run Step 4

If you have not already done so, complete the model in the `Project scheduling STEP 4` model file and run the model. If you are having any problems, import and look at the solution in the `Project scheduling SOL 4` project, available in the `Solutions` directory, to fix any errors in your model.

In the **Problem browser**, scroll down to the decision variables and hover the mouse on a variable name to see the icon for displaying the data view. Click this icon for each variable to view the solution. Specifically, look at the **task** variable to see which tasks are present in the schedule. Also, look at the **makespan** and notice that it has been reduced significantly compared to the solution from Step 3. This is due to the tradeoff between minimizing the makespan and minimizing the non-execution cost.

The final OPL model

```
using CP;

/********
 * Data *
 ********/

tuple Task {
 key int id;
 string name;
 int ptMin; //minimum duration
};
{ Task } Tasks = ...;
int TopLevelTask = ...;

tuple ParentLink {
 int taskId;
 int parentId;
};
{ ParentLink } ParentLinks = ...; // set for task hierarchy
{ int } Parents = { p.parentId | p in ParentLinks };

tuple Precedence {
 int beforeId;
 int afterId;
 string type;
 int delay;
};
{ Precedence } Precedences = ...;

tuple Worker {
 key int id;
 string name;
 float fixedCost;
 float propCost;
};
{ Worker } Workers = ...;
```

```
tuple Skill {
 key int id;
 string name;
};
{ Skill } Skills = ...;

tuple Proficiency {
 int workerId;
 int skillId;
 int level;
};
{ Proficiency } Proficiencies = ...;

tuple Requirement {
 key int id;
 int taskId;
};
{ Requirement } Requirements = ...;

tuple RequiredSkill {
 int reqId;
 int skillId;
 int levelMin;
 int levelMax;
};
{ RequiredSkill } RequiredSkills = ...;

{ int } PossibleWorkers[r in Requirements] =
 { p.workerId | p in Proficiencies, n in RequiredSkills :
 (n.reqId==r.id) &&
 (p.skillId==n.skillId) &&
 (n.levelMin <= p.level) &&
 (p.level <= n.levelMax) };

tuple Alloc {
 int reqId;
 int workerId;
 int pt;
};
{ Alloc } Allocations = { <r.id, i, t.ptMin> | r in Requirements, t in Tasks, i in
PossibleWorkers[r] : t.id==r.taskId };
```

```
tuple WorkerBreak {
 int workerId;
 int start;
 int end;
};
{ WorkerBreak } WorkerBreaks = ...;

tuple NonMandatoryTask {
 int taskId;
 float nonExecCost;
};
{NonMandatoryTask} NonMandatoryTasks = ...;

{ int } NonMandatoryTaskIds = { n.taskId | n in NonMandatoryTasks }; // set of non-manda-
tory task ids

// KPIs
float MakespanWeight = ...;
float FixedWeight = ...;
float ProportionalWeight = ...;
float NonExecutionWeight = ...;

int MaxInterval = (maxint div 2)-1;

tuple Step {
 int v;
 key int x; // Used to sort
};
sorted {Step} Steps[w in Workers] =
 { <100, b.start> | b in WorkerBreaks : b.workerId==w.id } union
 { <0, b.end> | b in WorkerBreaks : b.workerId==w.id };

stepFunction calendar[w in Workers] = stepwise (s in Steps[w]) { s.v -> s.x; 100 };

/**********************
 * Decision variables *
 **********************/

dvar interval task[t in Tasks] optional size t.ptMin..MaxInterval;
dvar interval alts[a in Allocations] optional size a.pt..MaxInterval
  intensity calendar[<a.workerId>];
dvar interval workerSpan[w in Workers] optional;
```

```
/************************
 * Decision expressions *
 ************************/

dexpr int makespan = max(t in Tasks) endOf(task[t]);
dexpr int workerPropCost[w in Workers] = sum
   (a in Allocations: a.workerId==w.id) sizeOf(alts[a],0);
dexpr float workersFixedCost = sum(w in Workers) w.fixedCost *
   presenceOf(workerSpan[w]);
dexpr float workersPropCost = sum(w in Workers)
   w.propCost*workerPropCost[w];
dexpr float nonExecCost = sum(n in NonMandatoryTasks) n.nonExecCost*(1-
   presenceOf(task[<n.taskId>]));

/*************
 * Objective *
 *************/

minimize MakespanWeight * makespan
 + FixedWeight * workersFixedCost
 + ProportionalWeight * workersPropCost
 + NonExecutionWeight * nonExecCost;

/***************
 * Constraints *
 ***************/

subject to {
 // Work breakdown structure
 forall (t in Tasks : t.id in Parents)
   span(task[t], all(i in ParentLinks: i.parentId == t.id)
   task[<i.taskId>]);

 // Precedence constraints
 forall (p in Precedences : p.type == "StartsAfterStart")
   startBeforeStart(task[<p.beforeId>], task[<p.afterId>], p.delay);
 forall (p in Precedences : p.type == "StartsAfterEnd")
   endBeforeStart(task[<p.beforeId>], task[<p.afterId>], p.delay);
```

```
 // Non-mandatory tasks in work breakdown structure
 presenceOf(task[<TopLevelTask>]); // top level task must be present
 forall(p in ParentLinks: !p.taskId in NonMandatoryTaskIds)
 // mandatory tasks forced to be present if their parent task is present
     presenceOf(task[<p.parentId>]) => presenceOf(task[<p.taskId>]);

 // Alternatives of workers who can fulfill task requirement
 //(each requirement must be filled by one worker)
 forall(r in Requirements)
   alternative(task[<r.taskId>], all(a in Allocations: a.reqId==r.id)
   alts[a]);

 // Calculate whether each worker is used in the project using span
 //constraint
 forall(w in Workers)
   span(workerSpan[w], all(a in Allocations: a.workerId==w.id) alts[a]);

 // A worker can fill only one task requirement at any point in time
 forall(w in Workers)
 noOverlap(all(a in Allocations: a.workerId==w.id) alts[a]);
};
```

Summary

This example demonstrated how to use CPLEX CP Optimizer to efficiently model and solve a basic project scheduling problem, and how the basic problem can be easily extended to incorporate additional business requirements such as calendars and non-mandatory tasks. The modeling patterns that were demonstrated in this example include:

- Work breakdown structures
- Precedence constraints
- Worker assignments and costs
- Worker calendars
- Non-mandatory tasks

Example 3: Transportation scheduling

This example addresses a transportation scheduling problem where a number of truck drivers are available to be scheduled to complete a set of driving tasks. There are various constraints related to the shifts each driver is able to work, due to the Hours-Of-Service (HOS) regulations specified by the Department of Transportation. The objective is to minimize the total number of shifts worked.

Demonstrated features

The light bulbs in Figure 3.1 highlight the main CPLEX CP Optimizer features used to model this problem. The features that are not covered in this or the preceding examples are grayed out.

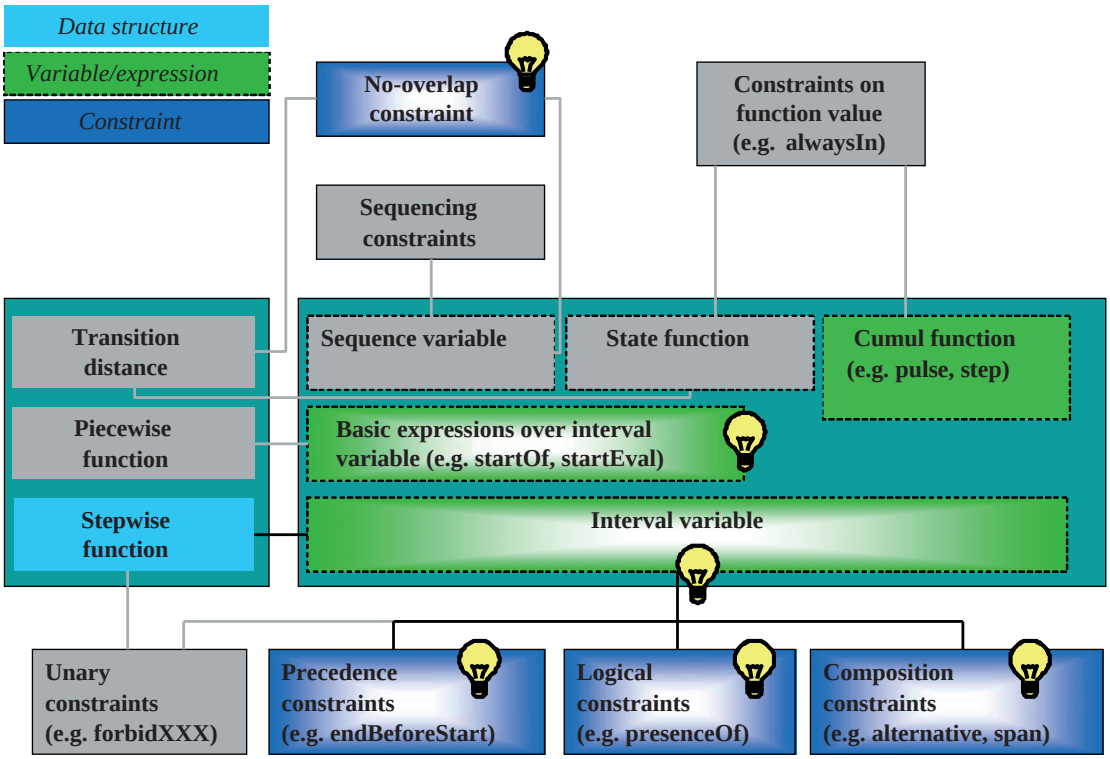


Figure 3.1: Modeling features demonstrated in the transportation scheduling example

The business problem

Eleven truck drivers are available to complete 100 driving tasks. Each task has a fixed duration and must be completed within a given time window. Each driver can work at most five shifts. The dates of shifts are not fixed, but shifts must obey the following hours-of-service rules:

- A shift cannot last more than 14 hours.
- For each shift, the driving time (the total duration of the tasks executed in this shift) cannot exceed 11 hours.
- There must be at least 10 hours of rest between two consecutive shifts.

Opening a shift for a driver incurs a fixed cost, and the objective is therefore to minimize the number of opened shifts.

To demonstrate how to model this problem, it is broken into the following steps:

- Step 1: Model the problem for a single driver
- Step 2: Extend the model to multiple drivers – an incomplete model
- Step 3: Extend the model to multiple drivers – the complete model

Each step includes descriptions of the data required to complete that part of the model, as well as the relevant definitions of the objective, decision variables, and constraints.

Step 1: Model the problem for a single driver

In order to consider this problem for a single driver, the problem description has to be modified slightly. Specifically, all tasks must be optional, because one driver cannot complete the same number of tasks originally intended for 11 drivers, and the objective is to maximize the total hours worked within the worker's five shifts.

The data

Tables 3.1 and 3.2 show representative data for this problem. All times are given in minutes. The corresponding data element names as used in the OPL projects are listed in brackets under the column names. The complete data set can be found in the data files included with the relevant projects (listed later in this example).

Table 3.1: Task data

Task id (`id`)	Earliest start time (`smin`)	Latest end time (`emax`)	Duration (`duration`)
1	696	3720	279
2	1013	3959	178
100	8361	9789	135

Table 3.2: Shift data

Data item	Value
Maximum number of shifts (`NbMaxShifts`)	5
Minimum rest between shifts (`MinInterShiftRest`)	600
Maximum work within a shift (`MaxIntraShiftWork`)	660
Maximum shift duration (`MaxShiftDuration`)	840

What is the objective?
The objective is to maximize the total hours worked. In other words, maximize the total duration of tasks completed by the worker within the five shifts.

What are the decisions to be made?
The decisions to be made are:

- The start time of each task
- The start time and duration of each shift
- Which tasks to complete in which shifts

What are the constraints?
The constraints to be incorporated in the model are:

- Each task can be scheduled on at most one shift.
- During each shift, the driver can complete only one task at a time.
- A shift cannot last more than 14 hours.
- For each shift, the driving time (the total duration of the tasks executed in this shift) cannot exceed 11 hours.
- There must be at least 10 hours of rest between two consecutive shifts.

OPL files
Use the following OPL projects:

- <*installDir*>/Steps/TransportScheduling/STEP1 SingleDriver/Transportation scheduling STEP 1: This OPL project is the starting point for completing the model. It includes all the data declarations and preprocessing required for this step, as well as comments to help you complete the objective, decision variable and constraint declarations.
- <*installDir*>/Solutions/TransportScheduling/STEP1 SingleDriver/Transportation scheduling SOL 1: The solution to Step 1.

<*installDir*> is the directory you downloaded the attached files to.

Take a look at the Transportation scheduling STEP 1 project in CPLEX Optimization Studio by doing the following:

1. Launch CPLEX Optimization Studio.

2. Import the *Transportation scheduling STEP 1* project by choosing **File** > **Import** > **Existing OPL projects**, selecting the <*installDir*>*/Steps/TransportScheduling/STEP1 SingleDriver* directory, and choosing the *Transportation scheduling STEP 1* project.

3. Take a look at both the model (.mod) and data (.dat) files to get a feel for the data representation and steps required to complete the model.

Each step in this example contains two run configurations: a default run configuration, as well as a run configuration that uses a reduced data set that can be used if you are working with a trial version of the software.

Model the decision variables

Consider the decisions to be made, listed earlier, to identify which decision variables (all of type **`interval`**) to declare.

The `interval` keyword

The OPL syntax for declaring an interval variable is as follows:

```
dvar interval a  [optional[(IsOptional)]] // For optional intervals
                 [in StartMin..EndMax] // Earliest start and latest end
                 [size SZ | in SZMin .. SZMax] // Duration
                 [intensity F]; // Intensity
```

More information can be found in the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL keywords > **interval**

Steps to complete the OPL model

1. Declare **`interval`** variables to represent:
    1. Each task
    2. Each shift
    3. Each possible task-shift combination. **Hint:** The set of all possible assignments of tasks to shifts has already been captured in the model, namely the **`Alternatives`** set.

2. Read through the problem description again and be sure to capture as many as possible relevant problem characteristics in the variable declarations.

The solution

The decision variables for this problem can be modeled using three arrays of **`interval`** variables. The first represents the tasks, and captures the earliest start, latest end, and duration of each task. These variables are declared optional for now, because one worker cannot complete all 100 tasks in the allotted time:

```
dvar interval task[t in Tasks] optional in
t.smin..t.emax size t.duration;
```

The second represents the shifts, and captures the maximum shift duration:

```
dvar interval shift[s in Shifts]
size 0..MaxShiftDuration;
```

The third represents all possible assignments of tasks to shifts. These variables are declared optional, because each task can be assigned to at most one shift:

```
dvar interval alt[a in Alternatives]
optional;
```

Model the objective

The objective is to maximize the total duration of tasks completed by the worker in the time available. The **`lengthOf`** keyword can be used for this purpose.

The `lengthOf` keyword

The expression **`lengthOf(a)`** returns the length of interval **`a`** if it is present in the schedule, where the length of a present interval is its end time minus its start time. If **`a`** is absent, the default value (0) is used. It is possible to specify a default other than 0.

For more information on the lengthOf keyword, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > **OPL** functions > **lengthOf**

Steps to complete the OPL model

Write an expression, using **`lengthOf`**, to maximize the total duration of scheduled tasks.

The solution

The objective is to maximize the total length of tasks included in the schedule. If a task is not executed, the value of the **`lengthOf`** expression for that task is 0.

```
maximize sum(t in Tasks) lengthOf(task[t]);
```

Note: For this particular example **lengthOf** and **sizeOf** can be used interchangeably. Because the intensity is at its default value, the length equals the size.

Model the constraints

Figure 3.2 shows an example of a possible solution for two shifts and six tasks. This figure represents all the constraints that must be captured in the model, including the relationships between the shift, alt, and task variables. The maximum shift duration has already been captured in the declaration of the shift variables. The dotted line around Task 6 indicates that this task is not included in the schedule.

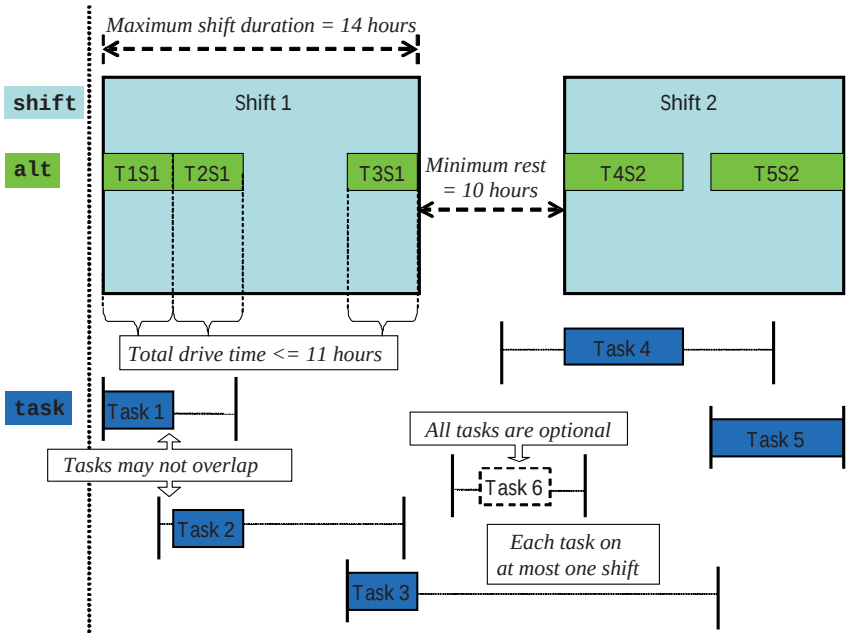


Figure 3.2: The constraints for the transportation model with a single driver

All the CPLEX CP Optimizer keywords required to model the constraints for this problem have been discussed in the preceding examples, and the overview of each keyword is therefore not repeated here. If you need to refresh your understanding of these keywords, see the overviews given in the preceding examples or the documentation as indicated next:

Precedence constraints
Example: *Project scheduling Step 1*
CPLEX Optimization Studio Help: Language > Language Reference Manual > OPL, the modeling language > Scheduling > **Precedence constraints between interval variables**

`noOverlap`
Example: *Project scheduling Step 2*
CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **noOverlap**

`span`
Example: *Project scheduling Step 1*
CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **span**

`alternative`
Example: *Oversubscribed scheduling*
CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **alternative**

Note that whereas the alternative constraint in the *Oversubscribed scheduling* example is a vertical alternative, the alternative in this example is a horizontal alternative. Specifically, the relationship between each task and the possible shifts it can be scheduled on can be expressed as a horizontal alternative. There are a discrete set of possible positions of the task in time, as represented by the **`alt`** variables and shown in Figure 3.2.

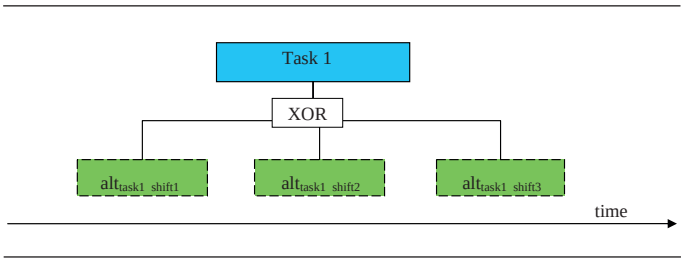


Figure 3.2: A horizontal alternative

Steps to complete the OPL model

After writing each constraint, you can choose to run the model and observe how the outcome changes with each additional constraint. Notice, for example, how the additions of the minimum rest time and maximum drive time constraints change the values of the **`shift`** variables. You can view variable values by clicking the icon next to the variable name in the CPLEX Studio **Problem browser.**

1. Use the **`alternative`** keyword to model the relationship between each task and the possible shifts it can be scheduled on. **Hint:** Use the **`alt`** variables.

2. Write a precedence constraint to model the 10 hours rest time required between consecutive shifts.

3. Write a **`noOverlap`** constraint to specify that each worker can complete at most one task at a time during each shift.

4. Write a **`span`** constraint to express the relationship between each shift and its associated alt variables.

5. Use **`lengthOf`** to express the constraint on the maximum driving time during each shift.

The solution

Use an **alternative** constraint to state that each scheduled task must use one of its alternatives, where each alternative is a task-shift combination. The **all** keyword is used to select only the **alt** variables associated with a particular task:

```
forall(t in Tasks)
  alternative(task[t], all(a in
  Alternatives: a.task==t) alt[a]);
```

To express the minimum rest time, use an **endBeforeStart** constraint to state that each shift must end before the start of the next shift, with a delay of **MinInterShiftRest** between shifts:

```
forall(s in 1..NbMaxShifts-1)
  endBeforeStart(shift[s], shift[s+1],
  MinInterShiftRest);
```

Use **noOverlap** to state that tasks within a given shift may not overlap. The **all** keyword is used to select the **alt** variables for a particular shift. The fact that shifts cannot overlap is already captured in the preceding precedence constraint.

```
forall(s in Shifts)
  noOverlap(all(a in Alternatives:
  a.shift==s) alt[a]);
```

The **span** constraint specifies that each shift spans its associated **alt** variables: each shift starts with the first **alt** interval for that shift and ends at the end of the last **alt** interval for that shift. The **all** keyword is used to select the **alt** variables for a particular shift.

```
forall(s in Shifts)
  span(shift[s], all(a in Alternatives:
  a.shift==s) alt[a]);
```

Finally, the **lengthOf** keyword is used to write the constraint on the maximum drive time within a shift. For each shift, the durations of all **alt** variables associated with the shift are summed up and required to be less than or equal to the maximum drive time:

```
forall(s in Shifts)
  sum(a in Alternatives: a.shift==s)
  lengthOf(alt[a]) <= MaxIntraShiftWork;
```

Note: If an **alt** variable is not present in the schedule, the **lengthOf** expression evaluates to zero

Run Step 1

If you have not already done so, complete the model in the `Transportation scheduling STEP 1` model file and run the model. If you are having any problems, import and look at the solution in the `Transportation scheduling SOL 1` project, available in the `Solutions` directory, to fix any errors in your model.

In the **Problem browser**, scroll down to the decision variables and hover the mouse on the variable names to see the icons for displaying the data view. Click these icons to view the solution. Click the **Scripting log** tab to see another view of the solution displayed according to the script at the end of the model (in the **execute** block). Notice that, with the 20 second solution time limit as indicated in the .ops file, the load on the first shift is 99%, and the load on the other four shifts is 100%.

In the next step, you will expand this model to include multiple drivers.

Step 2: Extend the model to multiple drivers – an incomplete model

The main characteristics to be incorporated into the model in order to extend it to multiple drivers are:

- 11 drivers are available.
- All tasks must be executed.
- Opening a shift for a driver incurs a fixed cost.
- The objective is to minimize the number of opened shifts.

All the shift constraints that applied to the single-driver model remain valid for multiple drivers.

The model presented in this step is not complete—there is an omission in the constraints that leads to an erroneous solution. If you reach the end of this step and know what the omission is, you can correct it before running the model. Otherwise, you can learn what the omission is and how to correct it in Step 3.

The data

The only additional data item required for this step is the number of drivers (see **`nbWorkers`** in the OPL .dat file). Also, the definition of the **`Alternative`** tuple changes to extend its use to multiple drivers. An **`Alternative`** is now a combination of a task, a driver, and a shift.

What is the objective?

The objective is to minimize the total number of opened shifts for all drivers.

What are the decisions to be made?

The decisions to be made are:

- The start time of each task
- The start time and duration of each shift for each driver
- The tasks to be completed by each driver in each of its shifts

What are the constraints?

All the constraints from Step 1 still apply, except that they must be extended to multiple drivers:

- Each task can be completed by at most one driver during one shift.
- Each driver can complete only one task at a time during each of its shifts.
- For each shift of each driver, the driving time (the total duration of the tasks executed in this shift) cannot exceed 11 hours.
- Each driver must get at least 10 hours of rest between two consecutive shifts.

OPL files

Use the following OPL projects:

- `<`*`installdir`*`>/Steps/TransportScheduling/STEP2 MultipleDriver Incomplete/Transportation scheduling STEP 2`: This OPL project is the starting point for completing the model. It includes all the data declarations and preprocessing required for this step, as well as comments to help you complete the model.
- `<`*`installdir`*`>/Solutions/TransportScheduling/STEP2 MultipleDriver Incomplete/Transportation scheduling SOL 2`: The solution to Step 2.

Take a look at the Transportation scheduling STEP 2 project in CPLEX Optimization Studio by doing the following:

1. Launch CPLEX Optimization Studio.

2. Import the `Transportation scheduling STEP 2` project by choosing **File** > **Import** > **Existing OPL projects**, selecting the `<`*`installdir`*`>/Steps/TransportScheduling/STEP2 MultipleDriver Incomplete` directory, and choosing the Transportation scheduling `STEP 2` project.

3. Take a look at both the model (.mod) and data (.dat) files to get a feel for the data representation and steps required.

Model the decision variables

The decision variables must be changed slightly to incorporate the following:

1. Tasks are no longer optional, because all tasks must be completed.

2. A total of 55 shifts (11 workers times five shifts) are available.

3. Shifts are now optional, because not all shifts are necessarily required to complete all the tasks.

The solution

The updated decision variable declarations are:

```
dvar interval task [t in Tasks] in t.smin..t.emax size t.duration;
dvar interval alt [a in Alternatives] optional;
dvar interval shift[w in Workers][s in Shifts] optional size 0..MaxShiftDuration;
```

Model the objective

The objective, in the case of multiple drivers with a fixed cost per shift, is to minimize the number of shifts used to schedule all the tasks. This can be modeled with the **`presenceOf`** keyword, which returns true if an interval is present in the schedule, and false otherwise.

For more information on **`presenceOf`**, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **presenceOf**

Steps to complete the OPL model

Use **`presenceOf`** to write the objective of minimizing the total number of shifts opened for all workers.

Steps to complete the OPL model

1. Change the variable declarations so a **`task`** is not optional, but a **`shift`** is.

2. Edit the declaration for **`shift`** so that the five shifts are available for each worker.

The solution

The objective is as follows:

```
minimize sum(w in Workers, s in Shifts)
presenceOf(shift[w][s]);
```

Model the constraints

All the constraints required for the single driver are still valid, except that they must be extended to multiple drivers where appropriate. The set of Alternatives has been modified by adding an additional dimension to handle multiple workers. Thus, the alternative constraint remains the same, because each task can be completed by at most one driver during at most one shift. The remaining constraints must be extended to multiple drivers, keeping in mind that the alt variables now represent combinations of drivers, shifts, and tasks, and that the shift variables are now indexed over both shifts and drivers.

Steps to complete the OPL model

Extend each of the constraints, except for the alternative constraint, to multiple drivers.

The solution

The set of constraints extended to multiple drivers is as follows:

```
forall (w in Workers, s in 1..
NbMaxShifts-1)
  endBeforeStart(shift[w][s], shift[w][s+1], MinInterShiftRest);

forall (w in Workers, s in Shifts) {

  span(shift[w][s], all(a in Alternatives: a.worker==w && a.shift==s) alt[a]);

  sum(a in Alternatives: a.worker==w && a.shift==s) lengthOf(alt[a]) <=
  MaxIntraShiftWork;

  noOverlap(all(a in Alternatives: a.worker==w && a.shift==s) alt[a]);

}
```

Run Step 2

If you have not already done so, complete the model in the `Transportation scheduling STEP 2` model file and run the model. If you are having any problems, import and look at the solution in the `Transportation scheduling SOL 2` project, available in the `Solutions` directory, to fix any errors in your model.

Click the **Scripting log** tab to see the solution displayed according to the script at the end of the model (in the **`execute`** block). At the end of the 5-minute time limit, the objective is 31 and the average shift load is 92%. This answer looks quite good, but there is something wrong with this solution. Look again at the values of the **`shift`** variables in the **Problem browser** and see whether you can find the error.

Because **`shift`** variables are now optional, it is possible to find solutions where the constraints on shifts, as extended from the single driver model with non-optional shifts, are not applied correctly. For example, the screenshot in Figure 3.3 shows a solution where driver 4 works only during shifts 2, 4, and 5. Because some shifts are not present and the shift numbers do not follow a chronological order, the precedence constraint that models the minimum rest time between subsequent shifts has been ignored. For example, the rest time between the end of shift 2 and the start of shift 5 is only 5 hours and 4 minutes, instead of 10 hours. Note that your solution may be slightly different due to differences in individual machine performance, and you may observe the same problem for other workers. In the next step, you will see how to correct this error.

Value for shift

Workers (size 11)	Shifts (size 5)	Interval			
		Present	Start	End	Size
1	1	0	0	0	0
1	2	0	0	0	0
1	3	1	0	703	703
1	4	1	4738	5578	840
1	5	1	7648	8488	840
2	1	1	2633	3473	840
2	2	0	0	0	0
2	3	1	6322	7016	694
2	4	0	0	0	0
2	5	1	0	653	653
3	1	0	0	0	0
3	2	0	0	0	0
3	3	1	1454	2294	840
3	4	0	0	0	0
3	5	0	0	0	0
4	1	0	0	0	0
4	2	1	5596	6436	840
4	3	0	0	0	0
4	4	1	796	1636	840
4	5	1	6740	7564	824

Figure 3.3: Incorrect solution—minimum rest time not captured

Step 3: Extend the model to multiple drivers – the complete model

The incomplete model lacks logic to force the numbering of the shifts, in the case where a driver works fewer than five shifts, to start at 1 and be incremented by 1 for each additional shift. For example, driver 4 in the incorrect solution works during shifts 2, 4, and 5, whereas these shifts should be numbered 1, 2, and 3. The model requires constraints that will force the optional shift variables to form a sequence for each worker, with an increment of 1 between two consecutive shifts in the sequence.

OPL files

Use the following OPL projects:

- `<installdir>/Steps/TransportScheduling/STEP3 MultipleDriver Complete/Transportation scheduling STEP 3`: This OPL project is the starting point for completing the model. It is equivalent to the solution to Step 2, and contains a comment to help you complete Step 3.
- `<installdir>/Steps/TransportScheduling/STEP3 MultipleDriver Complete/Transportation scheduling SOL 3`: The solution to Step 3.

Import the Transportation scheduling STEP 3 project by choosing **File** > **Import** > **Existing OPL projects**, selecting the <*installdir*>/Steps/TransportScheduling/STEP3 MultipleDriver Complete directory, and choosing the Transportation scheduling STEP 3 project.

Write constraints to enforce a chain of optional interval variables

A chain of optional interval variables can be enforced with CPLEX CP Optimizer by using a combination of precedence constraints and the presenceOf constraint. In general, for an array of optional interval variables, A, the following code will enforce a chain starting at A[1] (if present in the schedule) and with each subsequent present interval being incremented by 1:

```
dvar interval A[i in 1..n] optional …;
subject to{
  forall(i in 1..n-1) {
    endBeforeStart(A[i], A[i+1]);
    presenceOf(A[i+1]) => presenceOf(A[i]);
  }
};
```

The **endBeforeStart** precedence constraint forces the intervals to follow an ascending order. The second constraint states that the presence of interval **A[i+1]** implies the presence of interval **A[i]**. For example, if interval **A[4]** is present, then **A[3]** must also be present. The presence of **A[3]** in turn implies the presence of **A[2]**, which implies the presence of **A[1]**. Figure 3.4 shows such a chain of optional intervals:

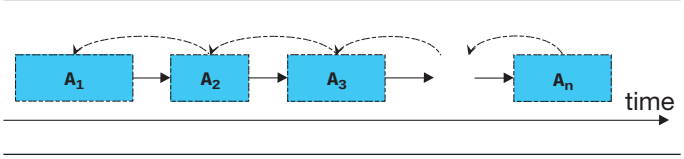


Figure 3.4: An optional interval chain

Steps to complete the OPL model

1. Look at the constraints given in the preceding paragraph and decide which of these are required to complete the model.

2. Add the appropriate set of constraints to create a chain of **shift** variables for each driver.

The solution

The precedence constraints to enforce a chain are already captured in the set of constraints that model the minimum rest time. Therefore, the only additional set of constraints required, is the one modeling the **presenceOf** implication between shifts for each driver:

```
forall (w in Workers, s in 1..
NbMaxShifts-1)
  presenceOf(shift[w][s+1]) =>
  presenceOf(shift[w][s]);
```

Run Step 3

If you have not already done so, complete the model in the Transportation scheduling STEP 3 model file and run the model. If you are having any problems, import and look at the solution in the Transportation scheduling SOL 3 project, available in the Solutions directory, to fix any errors in your model.

Click the **Scripting log** tab to see the solution displayed according to the script at the end of the model (in the **execute** block). At the end of the 5-minute time limit, the objective is now 34 and the average load per shift is 84%. Look again at the values of the **shift** variables in the **Problem browser** and see that the shifts for each worker form a chain starting at the first shift, and that the minimum rest time between shifts is respected.

The final OPL model

```
using CP;

/*******
* Data *
********

tuple Task {
 key int id;
 int smin;
 int emax;
 int duration;
};
{ Task } Tasks = ...;

int NbWorkers = ...;
int NbMaxShifts = ...;
int MinInterShiftRest = ...;
int MaxIntraShiftWork = ...;
int MaxShiftDuration = ...;

range Workers = 1..NbWorkers;
range Shifts = 1..NbMaxShifts;

// Alternatives for each task, worker, shift combination
tuple Alternative {
 Task task;
 int worker;
 int shift;
};
{ Alternative } Alternatives = { <t,w,s> | t in Tasks, w in Workers, s in Shifts };

/*********************
* Decision variables *
**********************/

dvar interval task [t in Tasks] in t.smin..t.emax size t.duration;
dvar interval alt [a in Alternatives] optional;
dvar interval shift[w in Workers][s in Shifts] optional size
      0..MaxShiftDuration;
```

```
/************
* Objective *
*************/

// Minimize total number of opened shifts
minimize sum(w in Workers, s in Shifts) presenceOf(shift[w][s]);

/**************
* Constraints *
***************/

subject to {
 // For each task, alternatives on Workers x Shifts
 forall (t in Tasks)
 alternative(task[t], all(a in Alternatives: a.task==t) alt[a]);

 // Shift sequence for each worker
 forall (w in Workers, s in 1..NbMaxShifts-1)
 endBeforeStart(shift[w][s], shift[w][s+1], MinInterShiftRest);

 // Shift optionality chain for each worker
 forall (w in Workers, s in 1..NbMaxShifts-1)
 presenceOf(shift[w][s+1]) => presenceOf(shift[w][s]);

 forall (w in Workers, s in Shifts) {
 // Shift spanning interval
 span(shift[w][s], all(a in Alternatives: a.worker==w && a.shift==s)
 alt[a]);
 // Max intra-shift work constraint
 sum(a in Alternatives: a.worker==w && a.shift==s) lengthOf(alt[a]) <=
 MaxIntraShiftWork;
 // Worker unary capacity during shift
 noOverlap(all(a in Alternatives: a.worker==w && a.shift==s) alt[a]);
 }
}
```

Summary

This example demonstrated how to use CPLEX CP Optimizer to efficiently model and solve a transportation scheduling problem involving complex constraints on driver shifts. The modeling patterns that were demonstrated in this example include:

- Precedence constraints
- Worker assignments and costs
- Shift constraints
- Chains of optional intervals

Example 4: Personal task scheduling

This example addresses the problem of personal task scheduling, and is based on an application described by Refanidis (2007). A number of tasks, some of which may be split into several parts, must be scheduled during a given time horizon. The problem characteristics to be addressed include precedence constraints, transition times, task preferences, and calendars.

Demonstrated features

The light bulbs in Figure 4.1 highlight the main CPLEX CP Optimizer features used to model this problem. The features that are not covered in this or the preceding examples are grayed out.

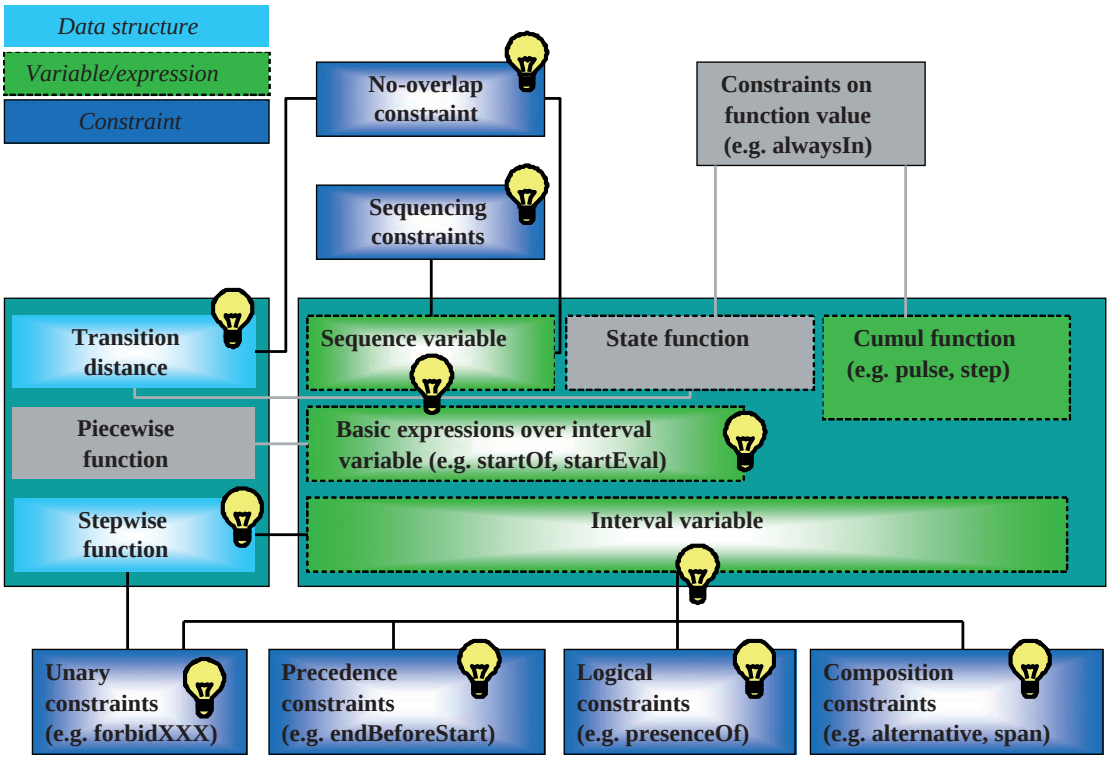


Figure 4.1: Modeling features demonstrated in the personal task scheduling example

The business problem

A number of personal tasks must be scheduled within a given time horizon. Some of these tasks may be split into several parts. Each task is associated with one of three locations, and all parts of a task must be completed at the task location. There is a transition distance between any two locations.

Tasks are characterized by:

- A fixed total processing time for each task
- A set of time windows (calendar) during which each task, or part of the task, may be scheduled
- Precedence constraints between tasks
- A satisfaction function for each task that calculates a measure of task satisfaction based on the schedule of the task parts

Task parts are characterized by:

- A minimum and maximum duration
- A minimum delay between consecutive parts of the same task

The objective is to maximize the total task satisfaction.

To demonstrate how to model this problem, it is broken into the following steps:

- Step 1: Model the tasks, task parts, and precedence constraints
- Step 2: Add the task part calendars and objective
- Step 3: Add the location transition distance matrix

Step 1: Model the tasks, task parts, and precedence constraints

You will write a basic personal task scheduling model without including calendars or transition distances.

The data

Tables 4.1 and 4.2 show representative data for this step. The time unit is hours and the total scheduling horizon is 500 hours. The corresponding data element names as used in the OPL projects are listed in brackets under the column names. The complete data set can be found in the data files included with the relevant projects (listed later in this example).

Table 4.1: Task data

Task id (`id`)	Task duration (`dur`)	Min part duration (`smin`)	Max part duration (`smax`)	Min delay between task parts (`dmin`)
1	8	2	5	8
2	7	7	7	0
5	7	2	5	8

Table 4.2: Precedence data

Predecessor task id (pred)	Successor task id (succ)
2	11
2	38
9	18

What is the objective?

The objective to maximize the total task satisfaction can only be added after the task part calendars have been modeled. In the meantime, an alternative objective is to minimize the makespan (the total time required to complete all the tasks).

What are the decisions to be made?

The decisions to be made are:

- The start time of each task
- The start time and duration of each task part

What are the constraints?

The constraints to be incorporated in the model are:

- The precedence constraints: Each task in the list of predecessor tasks must end before its successor may start.
- Each task must span its parts: A task must start with one of its parts and end with one of its parts.
- The sum of the durations of all the parts of a task must equal the task duration.
- The task parts used for each task must form a chain where the task part numbers must be in ascending order, with increments of 1.
- At most, one task part may be completed at a time.

OPL files

Use the following OPL projects:

- <*installdir*>/Steps/PersonalTaskScheduling/STEP1 PersonalTaskScheduling/Personal task scheduling STEP 1: This OPL project is the starting point for completing the model. It includes all the data declarations and preprocessing required for this step, as well as comments to help you complete the objective, decision variable and constraint declarations.
- <installdir>/Solutions/PersonalTaskScheduling/STEP1 PersonalTaskScheduling/Personal task scheduling SOL 1: The solution to Step 1.

<*installdir*> is the directory where you downloaded the attached files to.

Take a look at the Personal task scheduling STEP 1 project in CPLEX Optimization Studio by doing the following:

1. Launch CPLEX Optimization Studio.

2. Import the Personal task scheduling STEP 1 project by choosing **File** > **Import** > **Existing OPL projects**, selecting the <installdir>/Steps/PersonalTaskScheduling/STEP1 PersonalTaskScheduling directory, and choosing the Personal task scheduling STEP 1 project.

3. Take a look at both the model (.mod) and data (.dat) files to get a feel for the data representation and steps required to complete the model.

This model does not require a reduced data set for trial mode.

Model the decision variables

Consider the decisions to be made, listed earlier, to identify which decision variables (all of type **`interval`**) to declare.

The ***`interval`*** *keyword*

The OPL syntax for declaring an interval variable is as follows:

```
dvar interval a  [optional[(IsOptional)]] // For optional intervals
                 [in StartMin..EndMax] // Earliest start and latest end
                 [size SZ | in SZMin .. SZMax] // Duration
                 [intensity F]; // Intensity
```

More information can be found in the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL keywords > **interval**

Steps to complete the OPL model

Declare **`interval`** variables to represent the tasks and task parts. Be sure to capture as many as possible relevant problem characteristics in the variable declarations.

The solution

The decision variables for this problem can be modeled using two arrays of **`interval`** variables. The first represents the tasks, and captures the scheduling horizon:

```
dvar interval tasks[t in Tasks]
in 0..Horizon;
```

The second represents the task parts, and captures the minimum and maximum durations for each task part. The set of **`TaskParts`** is populated by calculating the maximum number of parts that a task can consist of assuming each part has a minimal duration. The **`parts`** variables are declared **`optional`**, because the maximum number of task parts will not necessarily be used for a task:

```
dvar interval parts[p in TaskParts]
optional size
 p.task.smin..p.task.smax;
```

Note: The task duration (**`task.dur`**) must not be used to declare the **`size`** of the **`tasks`** variables. The **`size`** of a **`tasks`** variable may be longer than the fixed duration of the associated task, because it will eventually be equal to the time between the start of the first task part and the end of the last task part.

Model the objective

The temporary objective for Step 1, until you have defined the satisfaction function, is to minimize the makespan. You can implement this using the **`endOf`** keyword, which returns an integer value equal to the end of an interval.

For more information on how to use **endOf** for this purpose, refer to the *Project scheduling* example, or to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **endOf**

Steps to complete the OPL model

Write an expression, using **endOf**, to minimize the latest completion time among all the task parts.

The solution

The objective can be written as follows:

```
minimize max(p in TaskParts)
endOf(parts[p]);
```

Model the constraints

Figure 4.2 summarizes the relationships between tasks and task parts to be captured in the constraints for Step 1.

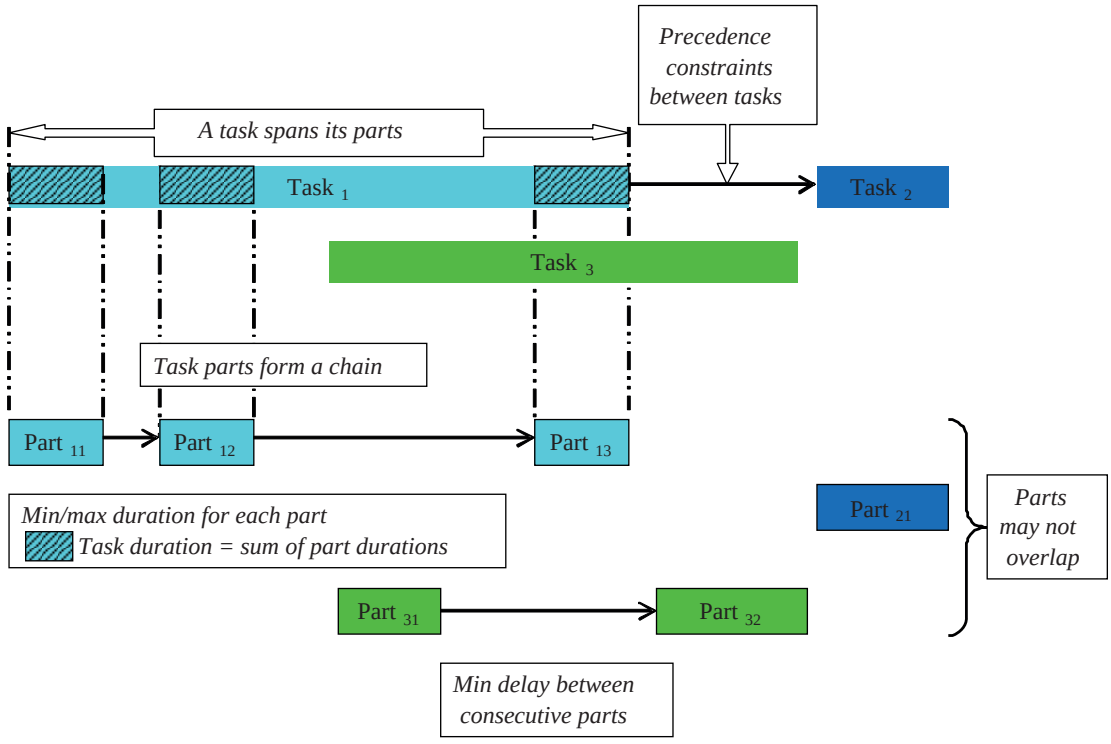


Figure 4.2: The relationships between tasks and task parts for Step 1

All the CPLEX CP Optimizer keywords required to model the constraints for this problem have been discussed in the preceding examples, and the overview of each keyword is therefore not repeated here. If you need to refresh your understanding of these keywords, see the overviews given in the preceding examples or the documentation as indicated next:

Precedence constraints
Example: *Project scheduling Step 1*
Documentation: Language > Language Reference Manual > OPL, the modeling language > Scheduling > **Precedence constraints between interval variables**

`noOverlap`
Example: *Project scheduling Step 2*
Documentation: Language > Language Quick Reference > OPL functions > **noOverlap**

`span`
Example: *Project scheduling Step 1*
Documentation: Language > Language Quick Reference > OPL functions > **span**

`sizeOf`
Example: *Project scheduling Step 2*
Documentation: Language > Language Quick Reference > OPL functions > **sizeOf**

`presenceOf`
Example: *Oversubscribed scheduling*
Documentation: Language > Language Quick Reference > OPL functions > **presenceOf**

Steps to complete the OPL model

1. Write a set of constraints to model the task precedence based on the set of **`Orderings`**.

2. For all tasks, write a constraint that states that the duration of a task must equal the sum of durations of its task parts. **Hint:** Use the **`sizeOf`** keyword.

3. For all tasks, write a constraint that states that each task must span its parts.

4. Write a set of constraints to specify that the task parts chosen for each task must form a chain where the parts present in the schedule must be numbered in ascending order, with increments of 1. **Hint:** Use the same modeling pattern as demonstrated in Step 3 of the *Transportation scheduling* example.

5. Incorporate the minimum delay between task parts in the constraints you just completed.

6. Write a **`noOverlap`** constraint to specify that at most one task part can be completed at any point in time.

The solution

Use a set of **endBeforeStart** constraints to model the task precedences:

```
forall(o in Orderings)
  endBeforeStart(tasks[<o.pred>], tasks[<o.succ>]);
```

The following set of constraints state that for each task, the duration of the task must equal the sum of durations of its task parts, and that each task must span its parts. The **sizeOf** keyword is used to determine the duration of each task part. Note that the **sizeOf** expression evaluates to 0 for any task parts not present in the schedule. Also note that because all tasks are compulsory, the first constraint implies that each task will consist of at least one part.

```
forall(t in Tasks){
 sum(p in TaskParts: p.task==t) sizeOf(parts[p]) == t.dur;
 span(tasks[t], all(p in TaskParts: p.task==t) parts[p]);
}
```

The chain of optional **parts** variables is modeled using the same modeling pattern as applied in *Step 3* of the *Transportation scheduling* example. The minimum delay between task parts is captured in the last argument of the **endBeforeStart** constraint:

```
forall(p in TaskParts){
 forall(s in TaskParts: s.task==p.task && s.id==p.id+1){
   endBeforeStart(parts[p], parts[s], p.task.dmin);
   presenceOf(parts[s]) => presenceOf(parts[p]);}}
```

Finally, a **noOverlap** constraint states that at most one task part may be completed at a time:

```
noOverlap(parts);
```

Note: The **noOverlap** constraint cannot be on the **tasks** variables, because task parts may be interleaved so that **tasks** intervals overlap even though the **parts** do not (see Figure 4.2).

Run Step 1

If you have not already done so, complete the model in the `Personal task scheduling STEP 1` model file and run the model. If you are having any problems, import and look at the solution in the `Personal task scheduling SOL 1` project, available in the `Solutions` directory, to fix any errors in your model.

Click the **Solutions** tab and see that the makespan found after the 1-minute time limit is around 353 (you may get a slightly different number depending on your computer's processing power). That is, all tasks have been scheduled to be completed within 353 hours. In the **Problem browser**, scroll down to the decision variables and hover the mouse on the variable names to see the icons for displaying the solution. Click these icons to view the solution. Look especially at the solution for the **`parts`** variables. Click the column titled **`task.id`** to sort according to increasing task numbers. Check that the **`parts`** for each task form a chain, and compare the **`size`**, start times and end times of the **`parts`** variables with that of the corresponding **`tasks`** variable to make sure your solution is correct.

In the next step, you will expand this model to include the task part calendars and the satisfaction functions.

Step 2: Add the task part calendars and objective

The basic model remains the same, except for the following changes:

- The objective to maximize the total task satisfaction replaces the Step 1 objective.
- Constraints are added to schedule task parts according to a preset calendar.

For the objective of maximizing task satisfaction, one of five types of satisfaction functions (see Figure 4.3) is associated with each task.

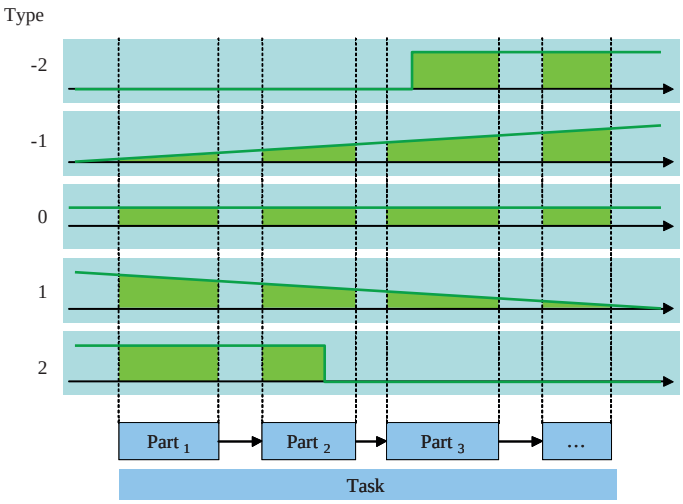


Figure 4.3: Five types of satisfaction functions

Type 0 is the simplest, where the task contributes a constant satisfaction value regardless of when its parts are scheduled. Type 2 is a step function that assigns a satisfaction value if a task part is scheduled before a given date and 0 otherwise. Type -2 works on the same principle, except that a value is assigned if a task is scheduled after a given date. Type 1 assigns increased satisfaction the earlier a task part is scheduled, while Type -1 assigns increased satisfaction the later a task part is scheduled.

For each task, a set of start dates and a set of end dates are specified to define a calendar. This calendar is used to declare the domain of a task. The domain is the set of time windows during which parts of that task may be scheduled, as shown in Figure 4.4.

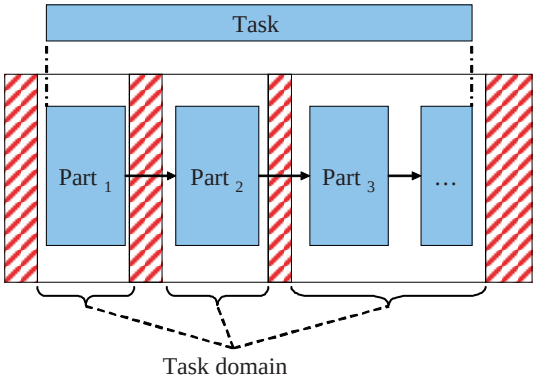


Figure 4.4: A task domain

The OPL statements that define the satisfaction functions and calendars are already completed in the OPL project for this step. You will use these definitions to complete the model.

The data

Table 4.3 shows a representative sample of the satisfaction function data required for this step. The date is required as input for the stepwise satisfaction function. Table 4.4 shows representative calendar data. The corresponding data element names as used in the OPL projects are listed in brackets under the column names. The complete data set can be found in the data files included with the relevant projects (listed later in this example).

Table 4.3: Satisfaction function data

Task id (`id`)	Satisfaction function type (`upref`)	Date (`date`)
1	0	0
11	-1	0
19	2	102

Table 4.4: Calendar data

Task id (`id`)	Domain start dates (`dstart`)	Domain end dates (`dend`)
1	20	29
1	42	57
1	70	86
2	24	41
2	36	53

OPL files

Use the following OPL projects:

- `<installdir>/Steps/PersonalTaskScheduling/STEP2 PersonalTaskScheduling/Personal task scheduling STEP 2:` This OPL project is the starting point for completing the model. It includes all the data declarations and preprocessing required for this step, as well as comments to help you complete the model.
- `<installdir>/Solutions/PersonalTaskScheduling/STEP2 PersonalTaskScheduling/Personal task scheduling SOL 2:` The solution to Step 2.

Take a look at the `Personal task scheduling STEP 2` project in CPLEX Optimization Studio. Look at both the model (.mod) and data (.dat) files to get a feel for the data representation and steps required to complete this step.

Model the objective

Steps to complete the model

1. Look at the definition of the **satisfaction** decision expression in the .mod file. If you are familiar with OPL syntax, try and understand how the different types of satisfaction functions are defined. Do not spend too much time on this if you are not familiar with OPL syntax.

2. Write the objective to maximize the total task satisfaction. Use the **satisfaction** decision expression.

The solution

```
maximize sum(t in Tasks) satisfaction[t];
```

Model the constraints

CPLEX CP Optimizer provides constraints to model restrictions that intervals cannot start, end, or overlap a set of time windows, where the set of time windows is represented by a stepwise function.

*The **forbidExtent** keyword*

Given an interval, **a**, and an integer stepwise function, **F**, the constraint **forbidExtent(a,F)** states that whenever the interval **a** is present, it cannot overlap a point in time, **t**, where **F(t)** = **0**, as shown in Figure 4.5.

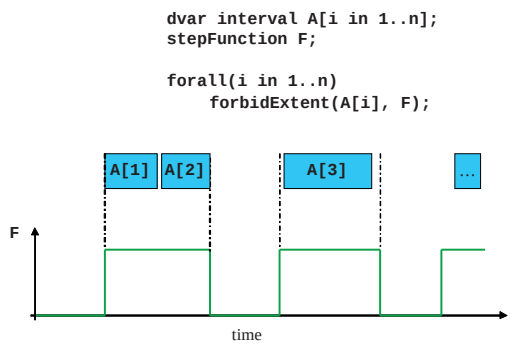


Figure 4.5: An example of the forbidExtent constraint

For more information on **forbidExtent**, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **forbidExtent**

Steps to complete the OPL model

Use **forbidExtent** to write a set of constraints that specify that a task part cannot overlap the time windows as specified in the **calendar** stepwise function for the associated task.

The solution

```
forall(p in TaskParts){
  forbidExtent(parts[p], calendar[p.task]);
```

Run Step 2

If you have not already done so, complete the model in the Personal task scheduling STEP 2 model file and run the model. If you are having any problems, import and look at the solution in the Personal task scheduling SOL 2 project, available in the Solutions directory, to fix any errors in your model.

In the **Problem browser**, scroll down to the decision variables and hover the mouse on the variable names to see the icons for displaying the data view. Click these icons to view the solution. In the view for the **parts** variables, click the **End** column to sort according to task part end time. Notice that the last task now ends much later than in Step 1 when task satisfaction and calendars were not considered. In the **Problem browser**, scroll down further to the decision expressions to see the values for the task satisfaction. The total task satisfaction after the 1-minute time limit is around 38. In the next step, you will expand this model to include a location transition distance matrix.

Step 3: Add the location transition distance matrix

The set of possible tasks must be completed at three different locations. Each task is associated with a location, and a transition distance is specified between each location pair. The transition distance corresponds to the transition time. The task parts must be scheduled in the original horizon of 500 days while considering the transition distances between the different locations.

With CPLEX CP Optimizer, you can easily apply transition times by passing a transition matrix as an argument to a **noOverlap** constraint. For this to work, the **noOverlap** constraint must be applied to a variable of type **sequence**. In this step, you'll learn more about this type of variable and how to use it, together with a **noOverlap** constraint and transition matrix, to model transition times.

The data

Tables 4.5 and 4.6 show a representative sample of the data required for this step. The corresponding data element names as used in the OPL projects are listed in brackets under the column names. The complete data set can be found in the data files included with the relevant projects (listed later in this example).

Table 4.5: Task location data

Task id (id)	Location (loc)
1	2
11	0
19	1

Table 4.6: Location transition matrix data

Location 1 (loc1)	Location 2 (loc2)	Transition distance (dist)
0	1	5
0	3	3

OPL files

Use the following OPL projects:

- `<installdir>/Steps/PersonalTaskScheduling/STEP3 PersonalTaskScheduling/Personal task scheduling STEP 3:` This OPL project is the starting point for completing the model. It includes all the data declarations and preprocessing required for this step, as well as comments to help you complete the model.
- `<installdir>/Solutions/PersonalTaskScheduling/STEP3 PersonalTaskScheduling/Personal task scheduling SOL 3:` The solution to Step 3.

Take a look at the `Personal task scheduling STEP 3` project in CPLEX Optimization Studio. Look at both the model (.mod) and data (.dat) files to get a feel for the data representation and steps required to complete this step.

Model the decision variables and constraints

A new variable of type **sequence** is required in order to pass the transition matrix as an argument to the **noOverlap** constraint.

The sequence decision variable

This type of decision variable defines a sequence of interval variables. The OPL syntax for declaring a **sequence** variable is as follows:

```
dvar sequence p in A [types T];
```

where **A** is an array of interval variables, and **T** is an array or set of integers, for example:

```
dvar interval A[];
int T[];
```

A sequence variable can be seen as an ordered set that contains as elements the interval variables from **A**.

Important: The order of the **`interval`** variables in the sequence does not necessarily correspond to the order of those **`interval`** variables in time. A **`noOverlap`** constraint must be applied to the **`sequence`** variable for the order of its **`interval`** variables to correspond to the order of those intervals in time. Other constraints, such as **`before, first, last`** and **`prev`**, are available to restrict the order of the intervals in a sequence, but these constraints do not have an effect on the order of the interval variables in time unless a **`noOverlap`** constraint is also applied to the sequence variable.

The optional array or set of (non-negative) integers, **`T`**, is used to associate a type with each interval variable in the sequence. When applying a transition matrix, the integers in **`T`** must correspond to the integers used for the row and column numbers of the matrix. The following example code shows how this is done for a set of tasks where a transition time must be applied. The data must include the set of tasks and the transition matrix, where the row and column numbers of the transition matrix correspond to an attribute of the task (in this case the task location):

```
tuple task{
 int id;
 int location;
};
{task} Tasks = …;

tuple transition{ int loc1; int loc2;
int value; };
{transition} Transitions = …;
```

Then, an array of interval variables, tasks, is defined over the set of **`Tasks`**:

```
dvar interval tasks[t in Tasks];
```

Next, a sequence variable, **`taskSequence`**, is defined to be an ordering of the **`tasks`** variables. The **`location`** of each task is specified as the type for each **`tasks`** variable in the sequence:

```
dvar sequence taskSequence in all(t in
Tasks)tasks[t] types all(t in Tasks)
Tasks[t].location;
```

Finally, a **`noOverlap`** constraint can be applied to the **`taskSequence`** variable, with the set of **`Transitions`** being an additional input. The effect of this constraint is that the **`tasks`** variables that are present in the schedule will not overlap in time, and the transition times as specified in the set of **`Transitions`** are respected:

```
noOverlap[taskSequence,Transitions];
```

For more information on the sequence variable, refer to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > **OPL keywords** > **sequence**

Steps to complete the model

1. Declare a **sequence** decision variable that will define an ordering of all task parts, and that uses the location as the type for each task part in the sequence.

2. Rewrite the **noOverlap** constraint from Step 2 to incorporate the **sequence** variable and the **Distances** transition matrix.

The solution

The **seq** variable defines a sequence of the **parts** variables, typed by the location:

```
dvar sequence seq in all(p in TaskParts)
parts[p] types all(p in TaskParts)
p.task.loc;
```

The **noOverlap** constraint on the **parts** from Step 2 is replaced by the following **noOverlap** constraint that incorporates the **seq** variable and the **Distances** transition matrix:

```
noOverlap(seq, Distances);
```

Run Step 3

If you have not already done so, complete the model in the *Personal task scheduling STEP 3* model file and run the model. If you are having any problems, import and look at the solution in the *Personal task scheduling SOL 3* project, available in the *Solutions* directory, to fix any errors in your model.

In the **Problem browser**, scroll down to the decision variables and hover the mouse on the variable names to see the icons for displaying the data view. Click these icons to view the solution. Look at the view for the **seq** variable to see the sequence in time in which the task parts will be executed. The name of each part displayed in this table (for example "parts#93") is generated automatically by OPL. To know which tasks these names correspond to, select the **Scripting log** where an alternative output of the solution is given. This log was generated by using the **PRINT_SEQUENCE** post-processing script block at the end of the .mod file. Notice the gap between the end time of one task and the start time of another task whenever there is a change in location.

In the view for the **parts** variables, click the **End** column to sort according to task part end time and see that the makespan was not affected much by the inclusion of transition distances. In the **Solutions** tab, see that the objective of total task satisfaction is slightly lower than before the transition matrix was incorporated.

The final OPL model

```
using CP;

/*******
* Data *
********/

int Horizon = 500;

tuple Task {
 key int id; // Unique identifier of the task
 int loc; // Location
 int dur; // Total duration
 int smin; // Minimum duration of a part
 int smax; // Maximum duration of a part
 int dmin; // Minimum delay between consecutive parts
 int upref; // Satisfaction function type
 int date; // Date used for stepwise satisfaction functions
 { int } dstart; // Start dates of domains (domains = time windows
 // during which task may be scheduled)
 { int } dend; // End dates of domains (domains = time windows during
 // which task may be scheduled)
};
{ Task } Tasks = ...;

int NbLocations = ...;

tuple Distance { int loc1; int loc2; int dist; };
{ Distance } Distances = ...; // transition distance matrix

tuple Ordering { int pred; int succ; }; // used for precedences
{ Ordering } Orderings = ...;

int DomMin [t in Tasks] = min(x in t.dstart) x;
int DomMax [t in Tasks] = max(x in t.dend) x;
int DomSize [t in Tasks] = DomMax[t]-DomMin[t];

tuple TaskPart {
 Task task;
 int id;
};
// task parts defined by number of minimal task parts that can fit in a // task
{ TaskPart } TaskParts = {<t,i> | t in Tasks, i in 1 .. t.dur div t.smin };
```

```
// steps to define calendar based on task domains
tuple Step { int x; int y; }
sorted { Step } Steps[t in Tasks] = { <x,0> | x in t.dstart } union { <x,100> | x in t.dend
};
// calendar based on task domains
stepFunction calendar[t in Tasks] = stepwise(s in Steps[t]) {s.y -> s.x; 0};

/*********************
* Decision variables *
*********************/

dvar interval tasks[t in Tasks] in 0..Horizon;
dvar interval parts[p in TaskParts] optional size
      p.task.smin..p.task.smax;
dvar sequence seq in all(p in TaskParts) parts[p] types all(p in
     TaskParts) p.task.loc;   // sequence of all task parts, typed by
                              // location

/************************
* Decision expressions *
************************/

dexpr float satisfaction[t in Tasks] = // task satisfaction function
 (t.upref==0) ? 1 :
   (1/t.dur)* sum(p in TaskParts: p.task==t)
   (t.upref==-2) ? maxl(endOf(parts[p]), t.date) –
   maxl(startOf(parts[p]), t.date) :
   (t.upref==-1) ? lengthOf(parts[p])*(DomMax[t] –
   (startOf(parts[p])+endOf(parts[p])-1)/2)/DomSize[t] :
   (t.upref== 1) ?
   lengthOf(parts[p])*((startOf(parts[p])+endOf(parts[p])-1)/2 –
   DomMin[t])/DomSize[t] :
   (t.upref== 2) ? minl(endOf(parts[p]), t.date) –
   minl(startOf(parts[p]), t.date) : 0;

/************
* Objective *
*************/

maximize sum(t in Tasks) satisfaction[t];
```

```
/**************
* Constraints *
***************/

subject to {

  // task precedence constraints
  forall(o in Orderings)
   endBeforeStart(tasks[<o.pred>], tasks[<o.succ>]);

   forall(t in Tasks) {
    // the sum of task part durations must equal total task duration
    sum(p in TaskParts: p.task==t) sizeOf(parts[p]) == t.dur;
    // task must span its parts
    span(tasks[t], all(p in TaskParts: p.task==t) parts[p]);
    }

  // task parts may only be scheduled to occur during task domains
  forall(p in TaskParts) {
   forbidExtent(parts[p], calendar[p.task]);

  // optional task part chain
  forall(s in TaskParts: s.task==p.task && s.id==p.id+1) {
    // task parts must follow an ascending order
    endBeforeStart(parts[p], parts[s], p.task.dmin);
    // presence of task part implies presence of subsequent task part
    presenceOf(parts[s]) => presenceOf(parts[p]);
   }
  }

  // task parts must not overlap, with transition distance matrix applied
  // according to type (location)
  noOverlap(seq, Distances);
  };
```

Summary

This example demonstrated how to use CPLEX CP Optimizer to efficiently model and solve a personal task scheduling problem that includes the possibility of breaking tasks into several parts. The model includes constraints on task precedence, calendars, transition times, and an objective of task satisfaction. The modeling patterns that were demonstrated in this example include:

- Precedence constraints
- Task calendars
- Optional task chains
- Transition times

Example 5: Batch scheduling

A set of product orders must be scheduled on a number of finite capacity machines. Each product is manufactured using a unique process that consists of multiple steps. In this example, the processing steps are heat treatments characterized by a combination of a temperature level and duration. Oven transition times between processing steps are dependent on the temperature change between steps. In addition, a material preparation time is required before each step of the same product order. The objective is to minimize the makespan.

Demonstrated features

The light bulbs in Figure 5.1 highlight the main CPLEX CP Optimizer features used to model this problem. The features that are not covered in this or the preceding examples are grayed out.

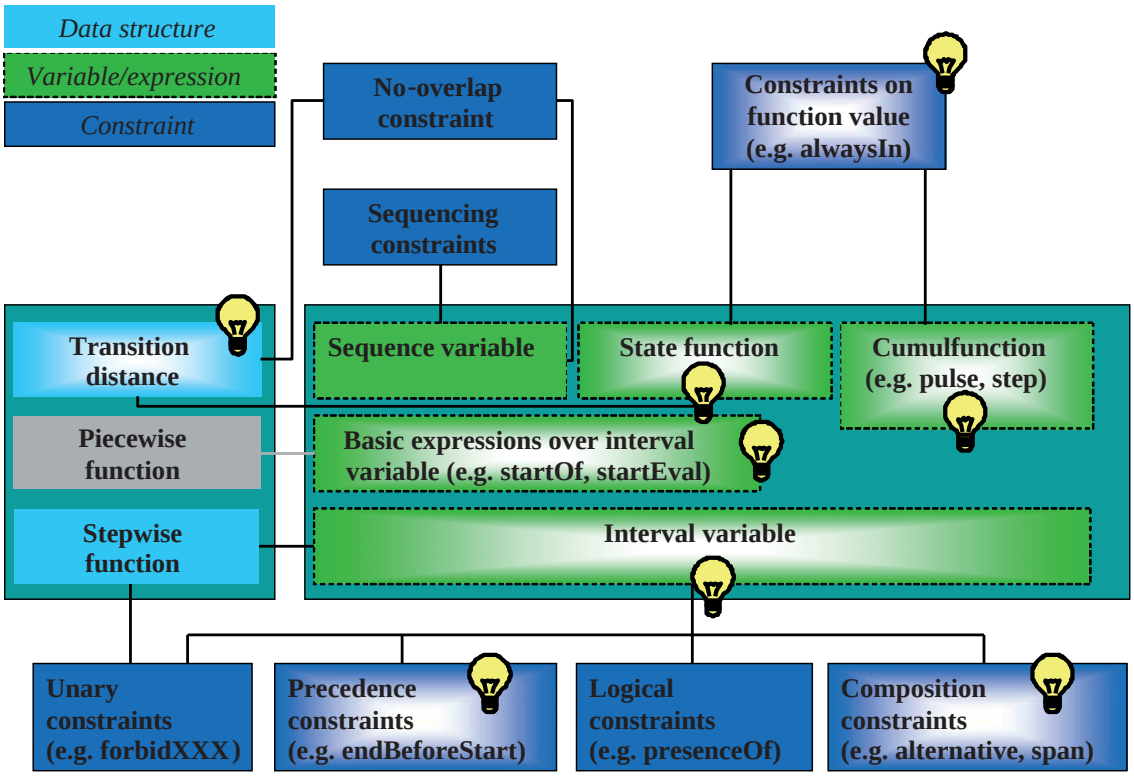


Figure 5.1: Modeling features demonstrated in the batch scheduling example

The business problem

A manufacturer produces five metallic alloys intended for the aerospace industry. Each alloy is manufactured using a unique heat treatment process consisting of up to four processing steps. Each processing step involves treating the alloy at one of four possible temperature levels for a fixed duration. The manufacturer has received a number of orders for each alloy product, and these orders must be scheduled on three ovens. Each oven can simultaneously process up to four orders at the same temperature.

The oven transition times between steps depend on the temperature difference between the steps. A material preparation time of 30 minutes is required between steps of the same product order, as well as before the first step of each order. This preparation time may overlap with the oven transition time. The goal is to find a production schedule that minimizes the makespan (the total time required to process all the orders). Figure 5.2 shows an example of this manufacturing process for three product, A (1 order), B (1 order), and C (2 orders), on two ovens, with each oven having the capacity to simultaneously process up to 2 orders at the same temperature.

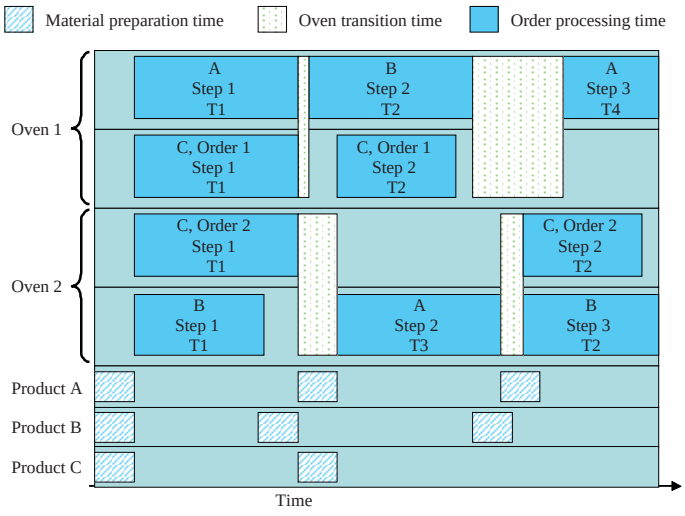


Figure 5.2: Example of the manufacturing process

To demonstrate how to model this problem, it is broken into the following steps:

- Step 1: Model the processing tasks assuming infinite capacity and ignoring temperature-dependent transition times
- Step 2: Add the oven capacity
- Step 3: Add the temperature-dependent transition times

OPL files

Use the following OPL projects:

- `<installdir>/Steps/BatchScheduling/STEPS BatchScheduling/Batch scheduling STEPS:` This OPL project contains four run configurations. Three run configurations correspond to each of the three steps, as indicated by the configuration number, and contain the relevant model, data, and setting files required as a starting point for completing that step. The remaining run configuration uses a trial data set with the Step 3 model and can be used if you have a trial installation of the software. Each model file contains all the data declarations and preprocessing required for the relevant step, as well as comments to help you complete the model.
- `<installdir>/Solutions/BatchScheduling/ SOLUTIONS BatchScheduling/Batch scheduling SOLUTIONS:` This OPL project also contains four run configurations: One for each of the three steps, as well as one for the solution to Step 3 with the reduced data set that can be run in trial mode.

`<installDir>`is the directory where you downloaded the attached files to.

Take a look at the Batch scheduling STEPS project in CPLEX Optimization Studio by doing the following:

1. Launch CPLEX Optimization Studio.

2. Import the Batch scheduling STEPS project by choosing **File** > **Import** > **Existing OPL projects**, selecting the <*installdir*>/Steps/BatchScheduling/STEPS BatchScheduling directory, and choosing the Batch scheduling STEPS project.

3. At the start of each step, take a look at the model (.mod) and data (.dat) files in the relevant run configuration (for example, the STEP 1 run configuration for Step 1) to get a feel for the data representation and steps required to complete the model.

Step 1: Model the processing tasks assuming infinite capacity and ignoring temperature-dependent transition times

In this step you do not have to consider the ovens, because of the assumption of infinite capacity. In addition, the temperature component of the processing can be ignored for now.

The data

Tables 5.1 and 5.2 show representative data for this step, where the time unit is minutes. The corresponding data element names as used in the OPL projects are listed in brackets under the column names. The complete data set can be found in the data files included with the relevant projects (listed later in this example).

Table 5.1: Processing step data

Product id (productId)	Step number (stepNumber)	Duration (duration)
1	1	30
1	2	30
1	3	30
2	1	45
2	2	45
2	3	45
2	4	45

Table 5.2: Order data

Product id (productId)	Number of orders (orders)
1	12
2	16

What is the objective?

The objective is to minimize the makespan (the total time required to complete all the orders).

What are the decisions to be made?

The decisions to be made are the start times of each processing task. A processing task is a combination of a product order and a processing step for that product.

What are the constraints?

The constraints to be incorporated in the model are:

- The precedence constraints: For each order, each processing step must finish before the next step in the process can start.
- A 30-minute material preparation time is required between steps of the same order, as well as before the very first step of each order.

Model the decision variables

The **`taskT`** tuple in the **`batchStep1`**.mod file defines a processing task as a combination of a product order and a processing step for that product. Declare decision variables of type **`interval`** to represent the processing tasks. Be sure to capture as many as possible problem characteristics in the declaration.

To see the syntax for declaring interval variables, refer to the Oversubscribed scheduling example, or to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL keywords > **interval**

The solution

The following array of **`interval`** variables represents the set of processing tasks. Each such interval must start no earlier than the initial 30 minutes preparation time, and end no later than some upper limit on the scheduling horizon (**`maxTime`**). The size attribute is equal to the duration of each task:

```
dvar interval tasks[t in allTasks] in
preparationTime..maxTime size t.duration;
```

Model the objective

Write an expression, using endOf, to minimize the latest completion time among all processing tasks.

For more information on how to use **`endOf`** for this purpose, refer to the *Project scheduling* example, or to the documentation available through CPLEX Optimization Studio Help: Language > Language Quick Reference > OPL functions > **endOf**

The solution

```
minimize max(t in allTasks) endOf(tasks[t]);
```

Model the constraints

The CPLEX CP Optimizer keywords required to model precedence constraints were presented in the *Project scheduling example (Step 1)*. You can also refer to the documentation available through CPLEX Optimization Studio Help at Language > Language Reference Manual > OPL, the modeling language > Scheduling > **Precedence constraints between interval variables**

Steps to complete the OPL model

1. Write a set of task precedence constraints for all tasks that belong to the same product and the same order.

2. Incorporate the material preparation time into the precedence constraints.

The solution

Use a set of **`endBeforeStart`** constraints to model the task precedence, while simultaneously capturing the preparation time between steps:

```
forall(t1,t2 in allTasks :  t1.productId == t2.productId &&
                            t1.stepNumber + 1 == t2.stepNumber &&
                            t1.orderId == t2.orderId)
                            { endBeforeStart(tasks[t1], tasks[t2],
                            preparationTime);
```

Run Step 1

If you have not already done so, complete the model in the `batchStep1.mod` file and run the `Batch STEP 1` run configuration by right-clicking the run configuration name and selecting **Run this**. If you are having any problems, import the `Batch scheduling SOLUTIONS` project, available in the Solutions directory, and look at the solution in the `batchSol1.mod` file to fix any errors in your model.

In the **Problem browser**, scroll down to the decision variables and hover the mouse on the **`tasks`** variable name to see the icon for displaying the solution. Click this icon to view the solution. Click twice on the column titled **Start** to sort according to earliest start time. Check that all tasks with **stepNumber = 1** start simultaneously (due to the unlimited capacity) at time 30, and that the 30-minute transition between tasks is consistently applied. Select the **Solutions** tab and see that the makespan is 300.

In the next step, you will expand this model to include the oven capacities.

Step 2: Add the oven capacity

To model the oven capacity, you must model the assignment of processing tasks to ovens and add constraints stating that the total number of processing tasks assigned to an oven at any point in time cannot exceed the oven capacity. The rest of the model remains the same as in Step 1.

The data

Table 5.3 shows the additional data required for this step. The corresponding data element names as used in the OPL projects are listed in brackets under the column names.

Table 5.3: Oven capacity data

Oven number (`ovens`)	Oven capacity (`ovenCapacity`)
1	4
2	4
3	4

What are the decisions to be made?

The additional decisions are the assignments of processing tasks to ovens.

What are the constraints?

The constraints to be incorporated are:

- Each processing task must be scheduled on one of the three ovens.
- Each processing task scheduled on an oven consumes one unit of oven capacity.
- For each oven, the oven usage must be less than or equal to the oven capacity at any point in time.

Model the decision variables

Declare decision variables of type **`interval`** to model the assignment of processing tasks to ovens.

The solution

The following array of **`interval`** variables represents all possible assignments of processing tasks to ovens. These variables are optional, because each processing task can be assigned to only one oven:

```
dvar interval taskOvenAlts[t in allTasks]
[o in ovens] optional;
```

Model the constraints

The oven capacity usage can be modeled using a cumulFunction.

The `cumulFunction` keyword

One of the uses of a cumulFunction is to model resource capacity. The value of a **`cumulfunction`** then represents the evolution of capacity over time, calculated from incremental changes (increases or decreases) by interval variables. The individual contribution of an interval to a **`cumulfunction`** is called an "elementary cumul function," for example the pulse function that was presented in the *Oversubscribed scheduling* example. Figure 5.3 shows an example of a composite cumul function for a resource where each task interval consumes one unit of capacity. The individual consumption by each task interval is modeled using the `pulse` elementary cumul function.

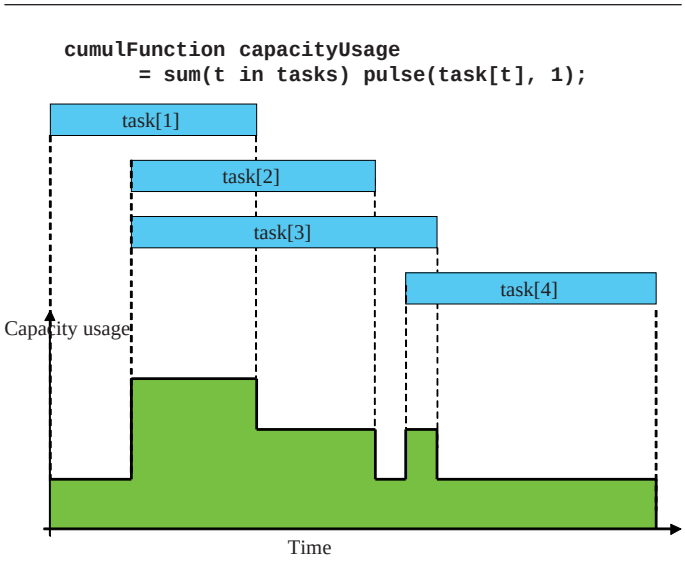


Figure 5.3: A composite cumul function

For more information on **`cumulFunction`** and elementary cumul functions, refer to the documentation available through CPLEX Optimization Studio Help at Language > Language Quick Reference > **OPL** keywords > **cumulFunction**

The constraint that each task must be scheduled on one of the three ovens is a vertical alternative and can be modeled using the **`alternative`** constraint. The **`alternative`** constraint was used in several of the preceding examples (such as the *Oversubscribed scheduling* example), and is therefore not explained here. You can also refer to the documentation available through CPLEX Optimization Studio Help at Language > Language Quick Reference > OPL **functions** > **alternative**

Steps to complete the OPL model

1. Write a **`cumulFunction`** to model the capacity usage of each oven.
2. Write a constraint to specify that, for each oven, the oven usage cannot exceed the oven capacity.
3. Use **`alternative`** to write a constraint stating that each processing task must be scheduled on only one of the ovens.

The solution

The **`ovenUsage cumulFunction`** represents the capacity consumption for each oven. **`pulse`** is used to model the contribution of each **`taskOvenAlts`** interval:

```
cumulFunction ovenUsage[o in ovens] =
   sum(t in allTasks)
  pulse(taskOvenAlts[t][o], 1);
```

Next, the **ovenUsage** is required to be less than or equal to the **ovenCapacity**:

```
forall(o in ovens)
   ovenUsage[o] <= ovenCapacity[o];
```

Finally, **alternative** is used to specify that each task must be scheduled on one of the three ovens:

```
forall(t in allTasks)
  alternative(tasks[t], all (o in ovens)
  taskOvenAlts[t][o]);
```

Run Step 2

If you have not already done so, complete the model in the `batchStep2.mod` file and run the `Batch STEP 2` run configuration by right-clicking the run configuration name and selecting **Run this**. If you are having any problems, import the `Batch scheduling SOLUTIONS` project, available in the Solutions directory, and look at the solution in the `batchSol2.mod` file to fix any errors in your model.

In the **Problem browser**, scroll down to the decision variables and view the solution for the **tasks** variables. Click the column entitled **Start** to sort according to start time, and notice that only 12 tasks are allowed to start at the earliest time so as not to exceed the total oven capacity. Look at the solution for the **taskOvenAlts** variables to see the assignment of tasks to ovens. In the Problem browser, scroll down further to the decision expressions to view the solution for the **ovenUsage cumulFunction**. Notice that each oven has usage 0 until the initial 30 minutes preparation is complete, and usage of 4 for the remainder of the scheduling horizon.

Select the **Solutions** tab and see that the makespan is 765. This is the optimal solution, because the shortest possible makespan (the total task duration divided by the total capacity, plus the initial 30-minute preparation time) is also 765.

This means that there is no idle time in the schedule due to the material preparation time, apart from the initial 30-minutes, and the ovens are fully utilized.

In the next step, you will expand this model to include the temperature-dependent transition times.

Step 3: Add the temperature-dependent transition times

Each processing step is associated with one of four temperature levels. An oven transition time applies whenever the oven has to change temperature between steps. You will learn how to add a **stateFunction** and an additional constraint to model these transition times, while the rest of the model remains unchanged.

The data

Tables 5.4 and 5.5 show representative data required for this step. The transition times are in minutes. The corresponding data element names as used in the OPL projects are listed in brackets under the column names.

Table 5.4: Processing step temperature data

Product (productId)	Processing step (stepNumber)	Temperature code (temperaturecode)
3	1	1
3	2	4
3	3	3

Table 5.5: Transition time data

Temperature code 1 (code1)	Temperature code 2 (code2)	Transition time (transitionTime)
1	1	0
1	2	5
1	3	10

What are the constraints?

The constraints must enforce the appropriate oven transition times when switching between processing steps with different temperatures.

Model the constraints

The `stateFunction` keyword

The purpose of a **`stateFunction`** is to model the change over time of the state of a resource, in this case the oven temperature. Changes in state often do not occur instantaneously, and transition times can be included in a **`stateFunction`** declaration to model the time required to change from one state to another. The general OPL syntax to declare a **`stateFunction`**, **`g`**, with a transition matrix, **`M`**, is as follows:

```
tuple triplet { int row; int column;
int value; };
{triplet} M = …;
stateFunction g with M;
```

The effect of the above declaration is that whenever the value of the **`stateFunction g`** changes, the appropriate transition time from the matrix **`M`** will be applied. For example if the matrix **`M`** has an entry {1; 2; 10}, then a transition time of 10 will be applied whenever the value of **`g`** changes from 1 to 2.

The value of a **`stateFunction`** is defined by constraints on the function that relate the function value to one or more **`interval`** variables. For example, the constraint **`alwaysEqual(f, a, v)`** specifies that over the interval a, the **`stateFunction f`** is always defined and equal to the value **`v`**, as shown in Figure 5.4.

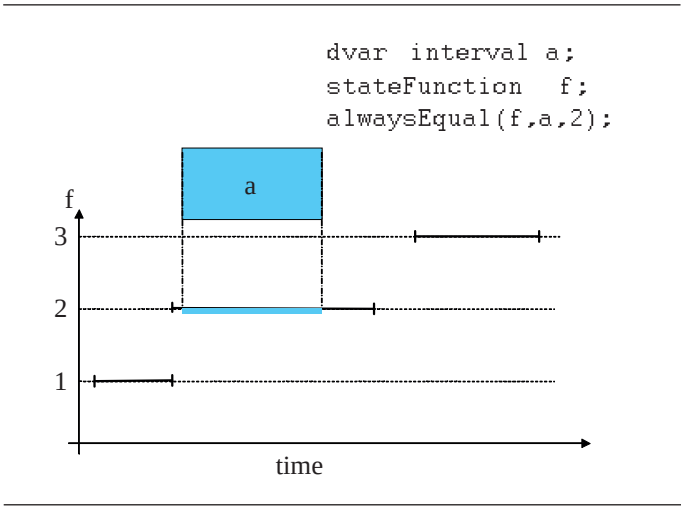


Figure 5.4: Example of a constraint on a stateFunction

Note: Unlike cumul functions, state functions are not decision expressions—they are not completely defined from a set of interval variables. The value of the state function will be decided by the solution engine so as to satisfy all the constraints on the state function. A state function behaves like a decision variable.

For more information on the **`stateFunction`** keyword, refer to the documentation available through CPLEX Optimization Studio Help at Language > Language Quick Reference > **OPL** keywords > **statefunction**

Steps to complete the OPL model

1. Declare a **`stateFunction`** called **`ovenTemperature`** to represent the temperature profile of each oven. Use the **`transitionTimes`** matrix.

2. Write a constraint to specify that, for each oven, the value of **`ovenTemperature`** at any point in time must equal the temperature code of the task scheduled on that oven.

The solution

The **ovenTemperature stateFunction** represents the temperature profile for each oven. The **transitionTimes** matrix is applied to model the time required between processing steps with different temperatures:

```
stateFunction ovenTemperature[o in ovens]
with transitionTimes;
```

The **alwaysEqual** constraint specifies that at any point in time, the value of **ovenTemperature** for an oven must equal the **temperatureCode** of the task scheduled on that oven:

```
forall(t in allTasks, o in ovens)
  alwaysEqual(ovenTemperature[o], taskOvenAlts[t][o],
  t.temperatureCode);
```

Run Step 3

If you have not already done so, complete the model in the `batchStep3.mod` file and run the `Batch STEP 3` run configuration by right-clicking the run configuration name and selecting **Run this**. If you are having any problems, import the `Batch scheduling SOLUTIONS` project, available in the Solutions directory, and look at the solution in the `batchSol3.mod` file to fix any errors in your model.

In the **Problem browser**, scroll down to the decision expressions to view the solution for the **ovenUsage cumulFunction**. Expand the value column as much as possible to get a better idea of the oven capacity usage. Notice that, unlike Step 2 where the ovens were 100% utilized after the initial preparation time, there is now significant variation in the oven capacity usage due to the temperature-dependent transition times. For the most part, periods of 100% usage (4 orders) are alternated with transition periods of 0% usage.

In the **Problem browser**, view the solution for the **ovenTemperature stateFunction** to see the temperature profile for each oven. A code of -1 is used where the function is undefined.

Select the **Solutions** tab and notice that the makespan is 800, which is slightly longer than the shortest possible makespan of 765.

The final OPL model

```
using CP;

/*******
* Data *
********/

int numberProducts = ...;
int preparationTime = ...;
range products = 1..numberProducts;

tuple productStepT {
 key int stepNumber;
 int duration;
 int temperatureCode;
};
{productStepT} steps[products] = ...;
```

```
int orders[p in products] = ...;

tuple taskT {
 key int productId;
 key int stepNumber;
 key int orderId;
 int duration;
 int temperatureCode;
};
{taskT} allTasks =  {<p, s.stepNumber, o,
                    s.duration, s.temperatureCode>
                    | p in products, s in steps[p], o in 1..orders[p]};

int numberOvens = ...;
range ovens = 1..numberOvens;

int ovenCapacity[ovens] = ...;

tuple transitionTimeT {
 key int code1;
 key int code2;
 int transitionTime;
};
{transitionTimeT} transitionTimes = ...;

int maxTime = sum(t in allTasks)(t.duration + preparationTime);

/*********************
* Decision variables *
*********************/

dvar interval tasks[t in allTasks] in preparationTime..maxTime size
   t.duration;
dvar interval taskOvenAlts[t in allTasks][o in ovens] optional;

cumulFunction ovenUsage[o in ovens] = sum(t in allTasks)
   pulse(taskOvenAlts[t][o], 1);
stateFunction ovenTemperature[o in ovens] with transitionTimes;
```

```
/************
* Objective *
*************/

minimize max(t in allTasks) endOf(tasks[t]);

/**************
* Constraints *
***************/

constraints {
 // preparation time between steps for each product order
 forall(t1,t2 in allTasks :  t1.productId == t2.productId &&
                            t1.stepNumber + 1 == t2.stepNumber &&
                            t1.orderId == t2.orderId) {
                            endBeforeStart(tasks[t1], tasks[t2], preparationTime);
                            }

 // for each oven, ovenUsage cannot exceed oven capacity
 forall(o in ovens)
   ovenUsage[o] <= ovenCapacity[o];

 // each task can be executed on at most one oven
 forall(t in allTasks)
   alternative(tasks[t], all (o in ovens) taskOvenAlts[t][o]);

 // the temperatureCode on an oven should correspond to the temperature of the task
scheduled on the oven at any point in time
 forall(t in allTasks, o in ovens)
   alwaysEqual(ovenTemperature[o], taskOvenAlts[t][o], t.temperatureCode);

}
```

Summary

You learned how to use CPLEX CP Optimizer to efficiently model and solve a batch scheduling problem that includes capacity constraints, setup times, and temperature-dependent transition times. The modeling patterns that were demonstrated in this example include:

- Precedence constraints
- State-dependent transition times
- Resource capacities
- Vertical alternatives

Conclusion

This paper demonstrated how to use CPLEX CP Optimizer to efficiently model and solve several practical scheduling problems, including:

- Oversubscribed scheduling (the satellite scheduling problem)
- Project scheduling
- Transportation scheduling
- Personal task scheduling
- Batch scheduling

It presented an overview of most of the features available in CPLEX CP Optimizer and showed how to use these features to write concise and simple models for complex real-world applications. In addition, it showed how you can use CPLEX Optimization Studio to implement CP models for an easy-to-use "model and run" approach.

References

L. Kramer, L. Barbulescu and S. Smith. "*Understanding Performance Tradeoffs in Algorithms for Solving Oversubscribed Scheduling*," Proc. AAAI-07, July, 2007.

I. Refanidis. "*Managing Personal Tasks with Time Constraints and Preferences*," Proc. ICAPS-07, September 2007.

Learn More

For more information on CPLEX CP Optimizer and scheduling, please visit http://www.ibm.com/software/integration/optimization/cplex-cp-optimizer/

For more information on IBM ILOG CPLEX Optimization Studio, please visit http://www.ibm.com/software/integration/optimization/cplex-optimization-studio/

For more information on CPLEX CP Optimizer and CPLEX Optimization Studio training, please visit http://www.ibm.com/software/websphere/education/curriculum/bpm/ILOG-Optimization.html or contact your IBM representative.

For more information

- For more information on IBM ILOG CP Optimizer and scheduling, please visit http://cpoptimizer.ilog.com
- For more information on IBM ILOG OPL, please visit http://oplstudio.ilog.com
- For more information on IBM ILOG CP Optimizer and IBM ILOG OPL training, please visit http://www.ilog.com/corporate/training or contact your IBM ILOG representative.

IBM Software Group
Route 100
Somers, NY 10589
U.S.A.

Produced in the United States of America
December 2010

Please Recycle

WSW14076-USEN-01